Luís Manuel da Costa Assunção

Mestre em Engenharia Informática

A Model for Scientific Workflows with Parallel and Distributed Computing

Dissertação para obtenção do Grau de Doutor em

Informática

Orientador: José Alberto Cardoso e Cunha,
Professor Catedrático (Aposentado),
Faculdade de Ciências e Tecnologia da
Universidade Nova de Lisboa

Júri

Presidente:	Luís C. Caires, Prof. Catedrático, FCT/Univ. Nova de Lisboa
Arguentes:	Fernando A. Silva, Prof. Catedrático, FC/Univ. do Porto
	Salvador Pinto Abreu, Prof. Catedrático, Univ. de Évora
Vogais:	José C. Cunha, Prof. Catedrático Aposent., FCT/Univ. Nova de Lisboa
	Manuel M. Barata, Prof. Coordenador, ISEL/Inst. Politécnico de Lisboa
	Pedro D. Medeiros, Prof. Associado, FCT/Univ. Nova de Lisboa
	João F. Sobral, Prof. Auxiliar, Escola de Eng. da Univ. do Minho
	Henrique João L. Domingos, Prof. Auxiliar, FCT/Univ. Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

dezembro, 2016

A Model for Scientific Workflows with Parallel and Distributed Computing

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

For all those who know how important they are in my life. In memory of my father and to my mother for continuing to be an example for me of never giving up when faced with the adversities of life.

(A todos os que sabem o quanto são importantes na minha vida. Em memória do meu pai e para a minha mãe por continuar a ser para mim um exemplo de nunca desisitir perante as adversidades da vida.)

"It does not matter how slowly you go as long as you do not stop."
Confucius

"I am a slow walker, but I never walk back."
Abraham Lincoln

"It always seems impossible until it's done."
Nelson Mandela

Acknowledgements

More than an academic requirement the journey to achieve the PhD degree was mainly decided as a way to improve my knowledge on the emergent paradigms related to parallel and distributing computing. The personal decision was taken after working fourteen years in informatics industry simultaneously with the main job as a teacher in Instituto Superior de Engenharia de Lisboa (ISEL). Therefore, the return as a student to the Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa was strongly a personal commitment.

However, during the long journey aiming at writing this dissertation, many individuals and institutions have played important roles that decisively contributed to make possible the achievement of this dissertation final results.

First of all, I am profoundly grateful to my supervisor, Professor José Cardoso e Cunha for his continuous support and commitment to make possible this dissertation. In fact, his constant availability for open discussions and scientific advising were fundamental to find directions and achieve solutions on many issues. Furthermore, without his human support and confidence on me, during moments of discouragement, this dissertation would not have been possible.

Secondly I would like to express my gratitude to my ISEL colleagues that directly or indirectly have contributed to make this dissertation possible. In particular, I would like to thank Carlos Gonçalves for his collaborative work and exchange of ideas and more important for relying on the AWARD prototype to perform experimental work in the context of his dissertation.

Several institutions were crucial to make easy and feasible my work. I would like to acknowledge the support (including financial contribution to expenses related to registration and travel to conferences abroad) of the following institutions: Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa (DI-FCT/UNL); Centro de Investigação (NOVA LINCS) of the DI-FCT/UNL; Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores of the Instituto Superior de Engenharia de Lisboa of the Instituto Politécnico de Lisboa (ADEETC-ISEL/IPL); Grupo de Investigação Aplicada em Tecnologias de Informação of the ADEETC-ISEL/IPL).

Faced with some health problems during the long journey I was led to change some behaviors of life. In fact, I lost a lot of weight, I quit smoking and I started walking almost

daily some Kilometers. However, I got help from three people who were fundamental.

I am profoundly grateful to Doctors Julia Toste and Paula Vasconcelos for their advising and success with me.

Graça, my wife and great marathon runner was and continues to be daily my guru contributing significantly for helping me to walk thousands of kilometers and enlightenment on my worst days.

Finally I consider friendship as an important factor in my life. Thus, in addition to my direct family I would like to thank Pedro, Sónia, Pacico, Rita, Vitor, Isabel, Nuno, Dina, Maria and Filomena for their long friendship and by continuing sharing with me some moments of their lives.

Abstract

In the last decade we witnessed an immense evolution of the computing infrastructures in terms of processing, storage and communication. On one hand, developments in hardware architectures have made it possible to run multiple virtual machines on a single physical machine. On the other hand, the increase of the available network communication bandwidth has enabled the widespread use of distributed computing infrastructures, for example based on clusters, grids and clouds. The above factors enabled different scientific communities to aim for the development and implementation of complex scientific applications possibly involving large amounts of data. However, due to their structural complexity, these applications require decomposition models to allow multiple tasks running in parallel and distributed environments.

The scientific workflow concept arises naturally as a way to model applications composed of multiple activities. In fact, in the past decades many initiatives have been undertaken to model application development using the workflow paradigm, both in the business and in scientific domains. However, despite such intensive efforts, current scientific workflow systems and tools still have limitations, which pose difficulties to the development of emerging large-scale, distributed and dynamic applications.

This dissertation proposes the AWARD model for scientific workflows with parallel and distributed computing. AWARD is an acronym for Autonomic Workflow Activities Reconfigurable and Dynamic.

The AWARD model has the following main characteristics.

It is based on a decentralized execution control model where multiple autonomic workflow activities interact by exchanging tokens through input and output ports. The activities can be executed separately in diverse computing environments, such as in a single computer or on multiple virtual machines running on distributed infrastructures, such as clusters and clouds.

It provides basic workflow patterns for parallel and distributed application decomposition and other useful patterns supporting feedback loops and load balancing. The model is suitable to express applications based on a finite or infinite number of iterations, thus allowing to model long-running workflows, which are typical in scientific experimention.

A distintive contribution of the AWARD model is the support for dynamic reconfiguration of long-running workflows. A dynamic reconfiguration allows to modify the structure of the workflow, for example, to introduce new activities, modify the connections between activity input and output ports. The activity behavior can also be modified, for example, by dynamically replacing the activity algorithm.

In addition to the proposal of a new workflow model, this dissertation presents the implementation of a fully functional software architecture that supports the AWARD model. The implemented prototype was used to validate and refine the model across multiple workflow scenarios whose usefulness has been demonstrated in practice clearly, through experimental results, demonstrating the advantages of the major characteristics and contributions of the AWARD model. The implemented prototype was also used to develop application cases, such as a workflow to support the implementation of the *MapReduce* model and a workflow to support a text mining application developed by an external user.

The extensive experimental work confirmed the adequacy of the AWARD model and its implementation for developing applications that exploit parallelism and distribution using the scientific workflows paradigm.

Keywords: Scientific workflows, Parallel and distributed computing, Dynamic reconfiguration.

Resumo

Assistimos na última década a uma imensa evolução das infraestruturas computacionais, tanto a nível de processamento, armazenamento e comunicação. Por um lado, os desenvolvimentos ao nível das arquiteturas hardware tornaram possível que se executem múltiplas máquinas virtuais sobre uma mesma máquina física. Por outro lado, o aumento da largura de banda disponível para comunicação em rede tornou possível a generalização do uso de infraestruturas computacionais distribuídas, por exemplo em *clusters*, *grids* e *clouds*. Estes fatores contribuíram decisivamente para que comunidades das mais variadas áreas da ciência pudessem almejar o desenvolvimento e execução de aplicações científicas complexas e com capacidade de processar grandes quantidades de dados. No entanto, pela sua complexidade estrutural, estas aplicações requerem modelos de decomposição em múltiplas tarefas executadas em paralelo e em ambientes distribuídos.

O conceito de *workflow* científico surge naturalmente como forma de modelar aplicações, por composição de múltiplas atividades. De fato nas últimas décadas múltiplas iniciativas foram empreendidas para usar o paradigma de *workflow* como forma de modelar o desenvolvimento de aplicações, tanto no mundo empresarial como em múltiplos domínios da ciência. No entanto, apesar do esforço intenso de múltiplas iniciativas, atualmente os sistemas e ferramentas associados aos *workflows* científicos ainda apresentam dificuldades no desenvolvimento das emergentes aplicações distribuídas dinâmicas e de larga escala.

Esta dissertação propõe o modelo AWARD, um acrónimo de *Autonomic Workflow Activities Reconfigurable and Dynamic*, para a área dos *workflows* científicos com computação paralela e distribuída.

O modelo AWARD tem as seguintes características principais.

O modelo é caracterizado por um modelo de controlo de execução descentralizado, em que as múltiplas atividades de um *workflow* são autónomas e comunicam entre si através de portas de entrada e saída. As atividades podem executar-se separadamente em ambientes computacionais diversificados, tais como num simples computador ou em múltiplas máquinas virtuais executadas em infraestruturas distribuídas, por exemplo em *clusters* e *clouds*.

Para a decomposição paralela e distribuída das aplicações, o modelo oferece os padrões básicos de *workflow* e outros padrões úteis, tais como, um padrão que suporta ciclos

de realimentação entre atividades (*feedback loops*) e um padrão de replicação de atividades para efeitos de equilíbrio de carga. O modelo também permite expressar aplicações baseadas num número finito ou infinito de iterações, o que suporta modelar e executar *workflows* de longa duração que são típicos na experimentação científica.

Uma contribuição distintiva do modelo AWARD é suportar a reconfiguração dinâmica durante a execução de *workflows* de longa duração. Uma reconfiguração dinâmica permite modificar a estrutura do *workflow*, por exemplo, introduzir novas atividades ou alterar as ligações entre as portas de entrada e saída. O comportamento das atividades de *workflow* pode também ser modificado, por exemplo, por substituição dinâmica dos algoritmos associados.

Para além de apresentar um novo modelo de *workflow*, esta dissertação apresenta a concretização de uma arquitetura de software totalmente funcional, que implementa o modelo proposto. O protótipo implementado serviu para validar e refinar o modelo através de múltiplos cenários de *workflow* cuja utilidade foi demonstrada na prática de forma clara, através de resultados experimentais, evidenciando as vantagens das principais características e contribuições do modelo AWARD. O protótipo implementado foi também usado para desenvolver casos de aplicação, tais como, um *workflow* que suporta a execução do modelo *MapReduce* e um *workflow* que suporta uma aplicação de análise de textos de língua natural, desenvolvida por um programador externo.

O extensivo trabalho de experimentação confirma a adequação do modelo AWARD e a operacionalidade da sua implementação para ser usado no desenvolvimento de aplicações que exploram o paralelismo e a distribuição baseando-se no paradigma de *workflows* científicos.

Palavras-chave: Workflows científicos, Computação paralela e distribuída, Reconfiguração dinâmica.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface.

AWA Autonomic Workflow Activity.

AWARD Autonomic Workflow Activities Reconfigurable and Dynamic.

BPEL Business Process Execution Language.

DAG Directed Acyclic Graph.

FCT-UNL Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa.

FIFO First-In, First-Out.

GUI Graphical User Interface.

HTTP Hyper Text Transfer Protocol.

OWL Ontology Web Language.

PN Process Networks.

PSE Problem Solving Environments.

RDF Resource Description Framework.

SPARQL Simple Protocol and RDF Query Language.

TCP/IP Transmission Control Protocol/Internet Protocol.

W3C World Wide Web Consortium.

WSDL Web Service Definition Language.

XML eXtensible Markup Language.

XSD XML Schema Definition.

Chapter 1

Introduction

Dissertation overview and the contributions achieved.

Diverse e-Science initiatives [FK03; Hin06; NeS12] have been requiring improved software tools in order to support more productive application modeling and experimentation in diverse application domains. This involves complex computational processes for simulation, visualization, access to large data sets, and increasing degrees of interaction with multiple users possibly located in different regions.

The workflow paradigm, early adopted in the business context [Hol95], is also useful for modeling scientific applications and allowing flexible mappings between the application abstractions and the underlying computing infrastructures namely large distributed infrastructures such as clusters, grids and clouds.

Workflows have been used for developing scientific applications in many domains [Chi+11; Dee+05; Tay+07; YB05]. Such efforts have been supported by multiple workflow tools [Laz11], as Triana [Tri11], Taverna [Tav11] and Kepler [Kep14]. However, currently, most of the existing scientific workflows tools do not offer adequate support to facilitate the development of several challenging scenarios that are becoming more important for emerging large-scale, distributed, and dynamic applications. As examples of currently open issues, existing workflow approaches still exhibit critical dependencies on a centralized enactment engine, and they lack flexibility in supporting the execution of long-running workflows with multiple iterations and in allowing their structural and behavioral dynamic reconfiguration.

This dissertation proposes the Autonomic Workflow Activities Reconfigurable and Dynamic (AWARD) model, describes its underlying support architecture as well as an implementation supported by a working prototype that has been used for executing concrete

workflow applications. The dissertation discusses how this approach provides feasible solutions to currently open issues as the ones mentioned above, and how this was achieved in practice by enabling a set of practical application scenarios. The experimentation that was conducted for validating the model and its implementation is also described.

The AWARD model is based on the Process Networks (PN) model of computation [Kah74] and it further defines the workflow activities (called Autonomic Workflow Activity (AWA)) as autonomic processes, which are logically distributed, have independent execution control, and can run in parallel on distributed infrastructures, such as clusters and clouds. Each AWA activity executes a software component, called *Task*, and developed as a Java class that implements a generic interface allowing programmers to code their applications without concerns for low-level details. In the AWARD model, the data-driven coordination of the interaction links between the AWA activities relies on a concept named AWARD Space, based on the Linda model [CG89]. The links between input and output ports of the AWA activities are abstractions supported by the AWARD Space as an unbounded and reliable global shared space.

In addition to supporting basic workflow patterns [Aal+00b] the AWARD model also supports non-basic patterns, such as feedback loops and load balancing contributing to more flexibility for developing scientific applications using workflows. Furthermore, the AWARD model supports dynamic workflow reconfigurations. During the execution of long-running workflows with multiple, possibly infinite, number of iterations the AWARD model supports the submission of reconfiguration plans as sequences of dynamic operator invocations, for structural and behavioral changes of one or more workflow activities. This characteristic of the AWARD model is a distinctive contribution of this dissertation towards increasing the flexibility to develop scientific workflow applications. In fact, many scientific experimental scenarios require continuous improvements without having to restart a new experiment each time some modification should be made to a workflow. In addition, by using dynamic reconfiguration plans, it is possibly to recover from failures and to steer workflows by multiple users. The coordination of the actions required by dynamic workflow reconfigurations uses the AWARD Space as an intermediary. The underlying AWARD architecture also uses the AWARD Space to support the monitoring of the execution of workflows, for instance, to detect failures.

This dissertation is not focused on workflow performance evaluation in the sense of program/application execution speed. However, the dissertation aims at contributing to increase the flexibility and efficiency in the process of solving real problems, by taking advantage of a user-friendly workflow model which addresses the main requirements for developing scientific applications. In addition, this dissertation provides a flexible and transparent implementation of an AWARD support framework decoupled from disparate technologies, thus promoting its reusability in multiple computing environments, for instance, for exploiting the parallelism and distribution currently available on computational infrastructures, namely clusters and clouds.

1.1 Motivation

The workflow paradigm offers a well established approach to deal with application complexity by supporting application decomposition into multiple activities. The workflow approach allows encapsulating parts of a problem within each activity, which can then be reused in different workflows for modeling different application scenarios. Furthermore, different forms of composition may easily be expressed in a workflow, namely specifying the sequential or the concurrent composition of activities, which opens the way to exploit parallelism and distribution in the workflow execution.

In addition, workflows facilitate large scientific experiments where different users with different expertise on scientific domains may develop specific tasks that can be combined to execute complex applications in parallel and distributed computing environments, possibly for processing large volumes of data.

Despite the multiple initiatives in scientific workflows, the scientific community has early realized that there were many open issues, as stated by multiple authors, for example, [Dee07; Dee+08]. However, many of these issues have remained open [Chi+11]. Also, in our early work in the context of the GeoInfo project at Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FCT-UNL) [Kaj+09] aiming at enabling large scale experimentation in Earth, Sea, and Space sciences, we found a need for improving the support provided by existing workflow tools in order to find solutions to many open issues [AGC09].

As a first issue, our preliminary work showed that many workflow approaches had little flexibility by requiring the scientific workflow developers to deal with low-level details of the execution engine. For example most existing workflow systems only support token data types compromised with the execution engine and have limitations for supporting new data types as application-dependent tokens. This can divert the developer's efforts from the application problem in many real-world scenarios. In fact, many scientific workflow developers come from diverse science domains without having a deep knowledge about computer science and technology.

As a result of our work and the survey conducted on a diversity of scientific publications on workflow systems, we have identified several open issues concerning both the high-level workflow abstractions and the workflow execution mechanisms, as summarized in the following:

- Lack of a decentralized execution control, leading to limitations on the workflow scale and on the benefits of using currently available parallel and distributed infrastructures;
- Lack of expressiveness for specifying the dependencies between the workflow activities, based on control-flow, data-flow, and their combinations;
- Lack of expressiveness for executing long-running workflows with multiple or infinite number of iterations as well as difficulties in expressing loops and feedback

dependencies among the workflow activities;

- Limitations to execute a workflow multiple times, as workflow instances, with parameter sweep;
- Lack of location transparency and dynamic binding for specifying the activity task. In most workflow systems the task development is critically coupled to the execution engine details or to the access to external services. For instance, Web services are statically defined not allowing dynamic bindings;
- Lack of support for exception and fault handling;
- Lack of an effective support for dynamic workflow reconfigurations during the execution of the multiple iterations of long-running workflows.

In the presence of the above issues and limitations we found motivation to propose the AWARD workflow model for overcoming the above difficulties and to implement a working prototype for supporting the practical development of scientific workflows.

1.2 Problem Statement

This dissertation addresses and proposes solutions to the following research question:

➦ How can a workflow model be designed and implemented in order to:

(A) Allow the development of scientific applications using the workflow paradigm, by requiring the programmer/developer to have only a minimum knowledge and concern about the low-level operational and implementation details of the underlying workflow enactment engine and execution environment;

(B) Support adequate and easy to use workflow abstractions, for expressing the application decomposition and coordination, and still allowing a transparent exploitation of parallel and distributed computing solutions, based on a model of decentralized control and distributed execution of the workflow activities;

(C) Enable the practical development of long-running experiments, by supporting the design and execution of workflows with multiple iterations, namely allowing workflows with a finite or infinite number of iterations, for example, to support data streaming;

(D) Support a flexible model for the structural and behavioral dynamic reconfiguration of workflows, by conciliating both its expressiveness (through a supported set of reconfiguration operators) and its feasible and effective practical implementation, in order to demonstrate its practical benefits, namely for dynamic reconfiguration of long-running workflows.

1.3 Requirements

Addressing all the above mentioned open issues would be a hard and complex task, and it would be impossible in a single dissertation. According to the (A), (B), (C), and (D) points of the above research question to guide the design of the AWARD model we identified the following main requirements that should be supported:

(A) a) Implementation of a workflow abstract machine for evaluating the feasibility of using the AWARD framework to develop concrete workflow application cases;

b) Transparency for specifying the application requirements without knowledge about low-level details of the execution engine. For instance, the control-flow or data-flow tokens between workflow activities should be only dependent on the application and not obscure data types tied to the execution engine details;

c) Decoupling the workflow specification and its mapping to the execution environments, for instance, the same workflow specification with minimal mappings should be: i) Executable on multiple standalone computers on a local network; or ii) Executed on multiple nodes of a cluster or on clouds using multiple virtual machines;

(B) a) Specification of workflows with expressiveness and flexibility for supporting multiple structural layouts including basic workflows patterns [Aal+00b] and non-basic workflow patterns, such as feedback loops between the workflow activities and activity replication for load balancing;

b) Autonomic behavior of each workflow activity for running without dependencies on a centralized execution engine in order to decentralize the control of parallel and distributed execution of workflows;

(C) a) Specification of long-running workflows with multiple iterations or even infinite number of iterations, where different activities can separately run in parallel by proceeding at their own pace in different iterations;

(D) a) Workflow reconfiguration by using a supported set of operators to dynamically change the structure and behavior of long-running workflows characterized by executing a large number of iterations;

b) The support of interactive workflows with user steering, where different users can execute, monitor and reconfigure the workflow activities;

Although we recognize their importance and the need for future research, the following issues are not addressed by this dissertation:

1. Description languages for specifying workflows;

2. Methods and tools for verification of correctness of the workflows. We assume that the correctness of dynamic workflow reconfiguration should be ensured by the workflow developer or by external tools according to the specific application scenario;

3. Development of user-friendly interfaces supporting the graphical design of the workflow specification mainly when the workflow has a great number of activities or even when multiple users are involved in the workflow development;

4. Tools for searching and dynamic binding of Web services for composing and orchestrating scientific workflows;

5. Scheduling the multiple workflow activities to multiple computational nodes;

6. Techniques for storing, accessing, sharing or moving large amounts of data;

7. Strategies for producing, storing and mining workflow provenance data.

1.4 Research Methodology

The methodology that guided this work encompassed the following steps:

1. Analyzing widely used workflow systems and tools, in order to characterize their advantages and identify open issues.

2. Outlining the requirements and designing the AWARD model to execute scientific workflows in order to achieve feasible and practical answers to the research question described in the above problem statement (Section 1.2);

3. Implementing the AWARD model, leading to a working prototype to allow real experiments on distributed infrastructures, such as networks of standalone computers, clusters and clouds;

4. Evaluating and refining the AWARD model and its implementation. Assessing the model expressiveness based on useful experimental scenarios and concrete application cases.

1.5 Dissertation Overview

During our preliminary work in the context of the GeoInfo project [Kaj+09] at FCT-UNL, which aimed at enabling large-scale experimentation in Earth, Sea, and Space sciences, we studied the development of workflows for composing scientific applications and we found a need for improving the support provided by existing workflow tools in order to find solutions to several open issues [AGC09]. Also, the survey conducted on related publications on workflow systems reinforced the need for investigating new directions in

order to improve the characteristics of scientific workflows including the possibility of applying dynamic reconfigurations to long-running workflows.

Furthermore, in the last decade many commercial initiatives and efforts by non-profit scientific communities contributed to easing the use of large-scale computing resources available on distributed infrastructures, such as clusters and clouds. Thus we found the need for decentralizing the control of the workflow execution in order to enable the opportunities for parallelism and distribution of the workflow activities.

The work carried out throughout this dissertation aimed at improving the functionality, flexibility and feasibility of the abstractions and mechanisms for specifying and executing long-running workflows, supporting their dynamic reconfiguration, and mappings to a diversity of computing infrastructures, from standalone computers to distributed clusters and clouds.

An overview of the main concerns addressed by this dissertation is presented in Figure 1.1.

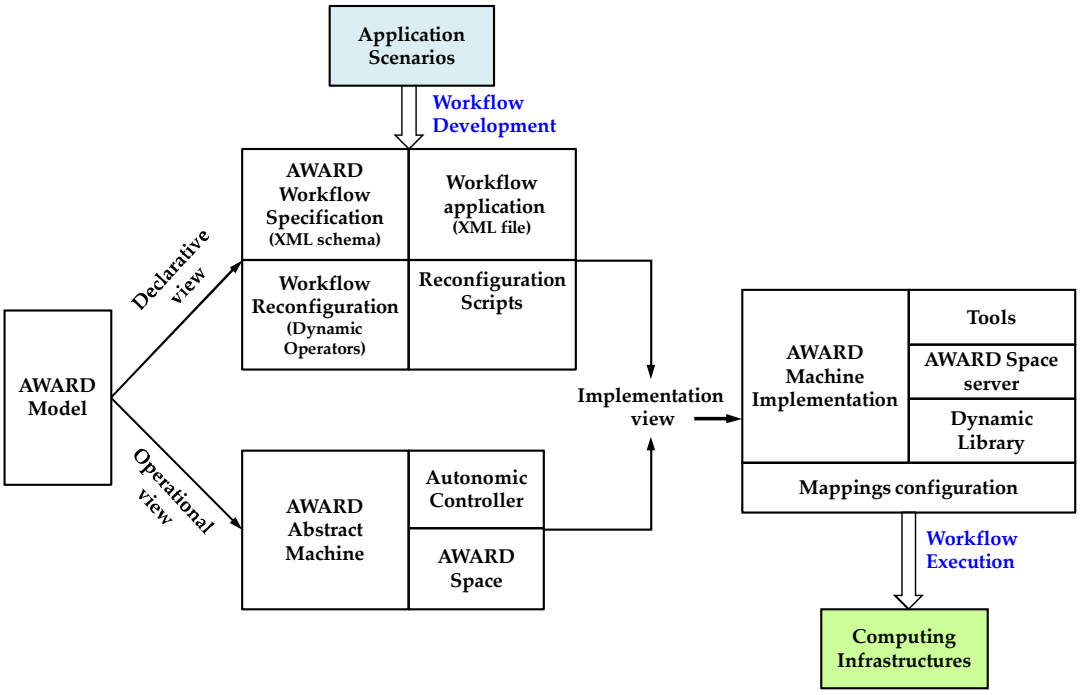


Figure 1.1: The dissertation concerns

A new scientific workflow model called AWARD, as an acronym for *Autonomic Workflow Activities Reconfigurable and Dynamic*, is proposed according to the requirements identified in the above Section 1.3.

The AWARD model is based on the Kahn Process Networks (PN) model of computation [Kah74] where processes communicate through unbounded First-In, First-Out (FIFO) channels by reading and writing atomic data elements called tokens. In the AWARD

model, a PN process is an Autonomic Workflow Activity (AWA) executed within an independent operating system process so it is possible that individual workflow activities or groups of activities are launched and executed on distributed computing nodes. In the AWARD model, the interaction between workflow activities is done through communication links, which are abstractions supported by the AWARD Space for carrying application-dependent tokens and connecting activity output ports to other activity input ports. Each autonomic workflow activity is enabled for executing its *Task* according to the tokens that have reached its input ports. In the AWARD model, the activity interaction is completely neutral with respect to the token data contents, which are application dependent and only interpreted by the activity *Tasks*.

The AWARD model is based on a decentralized control of the parallel and distributed execution of the workflow activities. The AWARD model supports basic workflow patterns [AHR11; Aal+00b], and non-basic workflow patterns, such as the feedback loop and the load balancing patterns.

The AWARD model supports structural and behavioral dynamic workflow reconfigurations to be applied to long-running workflows characterized by executing a large number of iterations. This important characteristic allows, for instance, recovering from faulty cloud services invoked by workflow activities [AC13]. Furthermore by combining the autonomic characteristics of the workflow activities and the support for dynamic reconfigurations, AWARD enables interactive workflows for user steering, where different users can execute, monitor and reconfigure the workflow activities [AC14].

As illustrated in Figure 1.1, the AWARD model is discussed according to three views described in the following:

1. The *Declarative view* represents the programmer's perspective used by developers to specify AWARD workflows for modeling their application scenarios, and preparing reconfiguration scripts in order to submit dynamic reconfiguration plans during the execution of the long-running workflows;

2. The *Operational view* describes the mechanisms of the AWARD abstract machine for executing AWARD workflows, and handling the requests for dynamic reconfiguration;

3. The *Implementation view* describes the implementation and architecture of the AWARD abstract machine as a framework composed of artifacts, such as tools, the AWARD Space server, and a software library (*DynamicLibrary.jar*) for supporting the mappings to computing infrastructures in order to execute and dynamically reconfigure the AWARD workflows.

The *Declarative view* encompasses the main concepts and formal definitions related to the specification of AWARD workflows, such as what is an activity and its input and output ports, what are links between output ports and input ports, what is a token passed through links, and what is an activity *Task*. This view also includes the definition of an

intermediate representation of AWARD workflows by relying on the standard eXtensible Markup Language (XML) schema language. This allows validating the conformance of the AWARD workflow specification using the XML language. Thus any editor of XML documents can be used to specify AWARD workflows. Additionally the concept of dynamic reconfigurations is described including the structure of any reconfiguration plan script based on a set of basic operators for supporting the submission of reconfiguration plans during the execution of the long-running workflows.

The *Operational view* describes the actions of the AWARD abstract machine for executing AWARD workflows and handling the dynamic reconfiguration requests. This abstract machine relies on two components: i) An *Autonomic Controller* with a *State Machine* and a *Rules Engine* for controlling the execution life-cycle of a workflow activity; and ii) The AWARD Space as an abstraction for supporting the links and the token passing between input and output ports of the workflow activities. This view includes the operational and semantics specification of the *Autonomic Controller* to be executed separately on any computing node, and the AWARD Space properties for supporting token passing and submission of dynamic reconfiguration requests.

The *implementation view* describes the implementation of the AWARD model through the AWARD framework as a set of artifacts allowing the workflow development and concrete mappings for executing scientific workflows on several computing environments, such as standalone computers, clusters and clouds. The architecture of the AWARD framework includes the following software components:

- A Java executable (*AwaExecutor.jar*) used for supporting the execution of an autonomic activity;
- A Java executable (*AwardSpace.jar*) used for launching a server, which implements the AWARD Space using the concept of tuple spaces, and also providing basic functionalities for logging and monitoring the workflow execution;
- A software library (*DynamicLibrary.jar*) allowing the development of dynamic reconfiguration scripts;
- A set of utility tools for launching one or more workflow activities in multiple computing nodes, namely in clusters and clouds, and for getting execution information on each workflow activity.

The decentralized control requirement, allowing each activity to have an autonomic behavior and to run on distributed computing nodes, increases the difficulty for monitoring and debugging the execution of long-running workflows. Our approach provides the basic mechanisms for logging information produced by all workflow activities to be used for monitoring and debugging the workflow execution.

Our work does not address the problem of how to assign the multiple workflow activities to a set of computational nodes, for instance cluster nodes or virtual machines on

clouds. This issue is left to the workflow developers. However, the AWARD framework tools for launching workflow activities can be used to assign the adequate computational nodes according to the application scenarios. For instance, developers can define partitions of activities and assign these partitions to the available computational nodes according to data location and other application requirements.

This dissertation presents an evaluation of the AWARD model and its implementation, including a comparison with other widely used scientific workflow systems, for instance, the Kepler workflow system. Three main dimensions are considered:

- Concerning parallel and distributed workflow execution, we evaluate the flexibility and functionality of the AWARD model to develop several workflow patterns and map their execution on parallel and distributed infrastructures;
- Concerning dynamic reconfiguration, we evaluate the AWARD model to support a set of useful scenarios, such as to change the activity tasks for increasing the performance or to recover from failures, and to change the workflow structure by introducing new activities, for instance, to introduce feedback loops or activity replicas for load balancing;
- Concerning the overall functionality provided by the AWARD framework, we evaluate its use for developing concrete application cases, such as a workflow for implementing the *MapReduce* model, a workflow where an activity *Task* invokes an external Web service, a workflow with steering by multiple users using dynamic reconfigurations, and a text mining application. We also evaluate the feasibility of the AWARD framework and its implementation in order to promote the transparent development of AWARD workflows by demonstrating that a user/developer is only required to know the AWARD declarative view and how to use the AWARD tools, without requiring a detailed knowledge about the AWARD machine internals.

1.6 Contributions

In the following the main contributions of this dissertation are identified:

Transparency: A workflow developer does not need to know low-level details of the execution engine for developing new workflow activities. The programming of the algorithm of a workflow activity (AWA *Task*) is similar to programming any Java desktop application which receives an array of invoking *Arguments*.

> **Contribution 1:** Workflow specification is decoupled from the execution environment being mostly focused on structural aspects and a declarative view of the logical dependencies between activities where the workflow developer defines: i) Names for activities; ii) Names for their input and output ports in order to establish the links to connect activities; iii) The token types associated to input and output ports;

and iv) The qualified name of the software components for implementing the algorithms of the workflow activities.

Decentralized control: The workflow execution is based on a decentralized control model, which allows the Autonomic Workflow Activities (AWA) to be launched and run separately on distributed computing nodes. Each AWA activity consumes and produces tokens at its own pace and can terminate independently of the others.

> **Contribution 2:** The workflow activities can be launched and run separately on heterogeneous infrastructures, ranging from a single computer to distributed infrastructures, such as network of local computers, clusters and clouds. Depending on the specific application scenarios a workflow can be subdivided into partitions with multiple activities. Each partition can be separately launched on different distributed sites and monitored by different users.

Dynamic reconfiguration: The structure and behavior of long-running workflows can be dynamically modified: i) By launching new activities; ii) By changing the activity parameters; iii) By changing the algorithm (AWA *Task*); iv) By creating input/output ports; v) By changing links between ports; vi) By changing the input and output ports behavior and the corresponding mappings to the AWA activity *Task Arguments* and *Results*;

> **Contribution 3:** A dynamic reconfiguration model relying on a set of primitive operators for providing a unifying approach to handle multiple aspects: i) Explicit dynamic workflow reconfiguration driven by users/tools on useful real scenarios; ii) Recovery from faults using dynamic reconfiguration operators; and iii) Workflow steering by multiple users, allowing each user to submit distinct reconfiguration plans.

Integration of the AWARD Space abstraction: The AWARD Space is a global shared tuple space based on the Linda model [CG89] used to support multiple aspects: i) The coordination of the interactions between AWA activities to exchange tokens related to data-flow or control-flow; ii) The submission of the dynamic workflow reconfigurations; and iii) Monitoring the execution of workflows allowing to analyze intermediate results, to observe the workflow state, the current elapsed execution times of each activity, and to detect possible activity failures;

> **Contribution 4:** The integration of the AWARD Space abstraction into the workflow model is completely transparent to the workflow developer in the following dimensions: i) The workflow developer only needs to specify the token types, which can be any according to the application requirements; ii) The workflow developer transparently invokes workflow reconfiguration plans through a software library (*DynamicLibrary.jar*), which automatically injects tuples in the AWARD Space representing reconfiguration scripts to affect multiple activities; and iii) The developer

benefits from the AWARD framework tools that provide transparent access to monitoring, inspection and debugging functionalities.

Implementation of a workflow prototype: The AWARD framework is supported by an effective architecture implementation composed of the following components: i) The kernel to control the life-cycle of an AWA activity; ii) The AWARD Space server; iii) The dynamic software library for applying dynamic reconfigurations; and iv) Tools to support the development and execution of AWARD workflows.

Contribution 5: The AWARD framework components were developed in Java and decoupled from particular computing environments. Their ease of use and portability allow the workflow development and execution on diverse computing infrastructures, such as a standalone computer, or virtual machines on distributed infrastructures, such as clusters and clouds.

1.7 Publications

The following publications describe the main results of our work based on the proposal of the AWARD model and its innovative contributions:

- Luís Assunção and José C. Cunha, "Enabling Global Experiments with Interactive Reconfiguration and Steering by Multiple Users," in Proceedings of International Conference on Computational Science (ICCS 2014) - Procedia Computer Science, pp. 2137-2144, Elsevier, 2014, Doi: 10.1016/j.procs.2014.05.198;

- Luís Assunção, Carlos Gonçalves, and José C. Cunha, "Autonomic Workflow Activities: The AWARD Framework," International Journal of Adaptive, Resilient, and Autonomic Systems (IJARAS), vol. 5(2), pp. 57-82, IGI Global, 2014, Doi: 10.4018/ijaras.2014040104;

- Luís Assunção and José C. Cunha, "Dynamic Workflow Reconfigurations for Recovering from Faulty Cloud Services," in IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom 2013), pp. 88–95, IEEE, 2013, Doi: 10.1109/CloudCom.2013.19;

- Carlos Gonçalves, Luís Assunção, and José C. Cunha, "Flexible MapReduce Workflows for Cloud Data Analytics," in International Journal of Grid and High Performance Computing (IJGHPC), vol. 5(4), pp. 48–64, IGI Global, 2013, Doi: 10.4018/ijghpc.2013100104;

- Carlos Gonçalves, Luís Assunção, and José C. Cunha, "Data Analytics in the Cloud with Flexible MapReduce Workflows," in IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom 2012), pp. 427–434, IEEE, 2012, Doi: 10.1109/CloudCom.2012.6427527;

- Luís Assunção, Carlos Gonçalves, and José C. Cunha, "Autonomic Activities in the Execution of Scientific Workflows: Evaluation of the AWARD Framework," in Proceedings of the 9th IEEE International Conference on Autonomic and Trusted Computing (ATC 2012), pp. 423-430, IEEE, 2012, Doi: 10.1109/UIC-ATC.2012.14;
- Luís Assunção, Carlos Gonçalves, and José C. Cunha, "On the Difficulties of Using Workflow Tools to Express Parallelism and Distribution - A Case Study in Geological Sciences," Proceedings of the International Workshop on Workflow Management of the International Conference on Grid and Pervasive Computing (GPC 2009). pp. 104–110, IEEE, 2009, Doi: 10.1109/GPC.2009.30.

1.8 Organization of the Dissertation

A global perspective of this dissertation is presented in Figure 1.2, including the main concerns discussed in each chapter regarding the development and evaluation of the AWARD model.

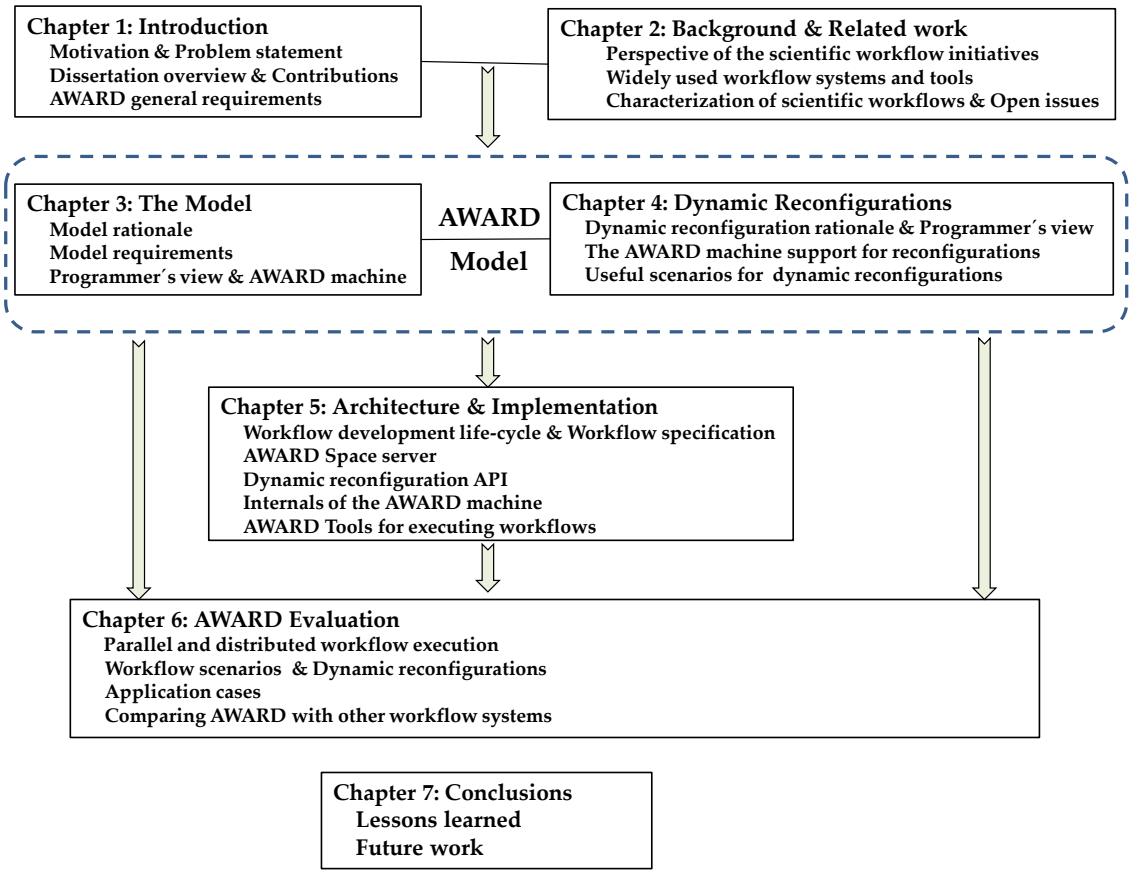


Figure 1.2: The dissertation chapters and the related main concerns

This dissertation is organized in seven chapters with the following contents. This Chapter (1) is dedicated to introduce the research questions, the motivations and the requirements behind the development of the AWARD model. This chapter also presents

the dissertation overview and discusses the main contributions, including the list of scientific publications produced during the accomplished research work.

In Chapter 2, background and related work on the scientific workflows area are outlined in order to identify important open issues that have motivated the specification of requirements to develop the AWARD model.

The remaining chapters present the AWARD model and its implementation in order to support the development of application scenarios based on the scientific workflow paradigm.

In Chapter 3, we discuss the rationale of the AWARD model, the concepts related to the specification of the AWARD workflows and the AWARD abstract machine that allows the parallel and distributed execution of the workflow activities, including the operational view of an *Autonomic Controller* for executing workflow activities.

In Chapter 4, we discuss how the AWARD abstract machine was extended in order to support structural and behavioral workflow dynamic reconfigurations. This includes the definition of a set of dynamic operators to be used in reconfiguration scripts.

In Chapter 5, we present the AWARD machine architecture and its implementation, including the internal components of the *Autonomic Controller*, such as the *State Machine* and the *Rules Engine*, and the AWARD Space server. A set of useful tools are described for launching and monitoring the execution of AWARD scientific workflows.

In Chapter 6, we present an evaluation of the AWARD model centered on the flexibility, functionality and feasibility for developing workflows. Three main dimensions are considered: Parallel and distributed execution; Support for dynamic workflow reconfigurations; Feasibility of using AWARD for implementing real application cases. Chapter 6 also compares the AWARD characteristics to other workflow systems.

Conclusions and future work directions are presented in Chapter 7.

CHAPTER 2

Background and Related Work

The background and related work, identifying issues that have motivated the development of the AWARD model.

This chapter discusses the background and related work in the scientific workflows area. After an introduction, in Section 2.1 for presenting a global perspective of the evolution of the scientific workflow paradigm, we describe, in Section 2.2, several widely used workflow systems, such as Pegasus, Triana, Taverna and Kepler.

In Section 2.3, we discuss the importance and the limitations that still exist for supporting replicability and reproducibility of scientific workflows.

In Section 2.4, we outline several dimensions of the scientific workflow characteristics. In section 2.4.1, we discuss generic characteristics, such as workflow structural patterns and specification languages, and the workflow computational models. In Section 2.4.2, we discuss the requirements and the existing limitations for distributed execution of workflows. In Section 2.4.3, we discuss the flexibility for supporting dynamic reconfigurations. Section 2.4.4 discusses the support for exception handling and fault recovery and, in Section 2.4.5, we discuss the support for interactive workflow steering by different users.

In Section 2.5 we present conclusions, and identify the requirements of the AWARD model.

2.1 Introduction

Technological improvements in computer hardware and communication networks that occurred in the last two decades have enabled the possibility of performing high throughput

computing on large sets of distributed computers. Previously, the execution of scientific applications with intensive processing requirements was only possible on supercomputers not easily available to many scientists or organizations.

The above mentioned technological improvements enabled the development of high performance scientific applications and their execution on clusters, grids or more recently on clouds.

Despite the availability of such distributed computing infrastructures the challenge remains how to perform application decomposition in multiple components and how to coordinate their execution on these infrastructures. Already successfully used during the past fifteen years in business process modeling, the workflow paradigm was also adopted for modeling scientific applications.

Figure 2.1 illustrates a global perspective of the scientific workflow initiatives and their relation to previous works.

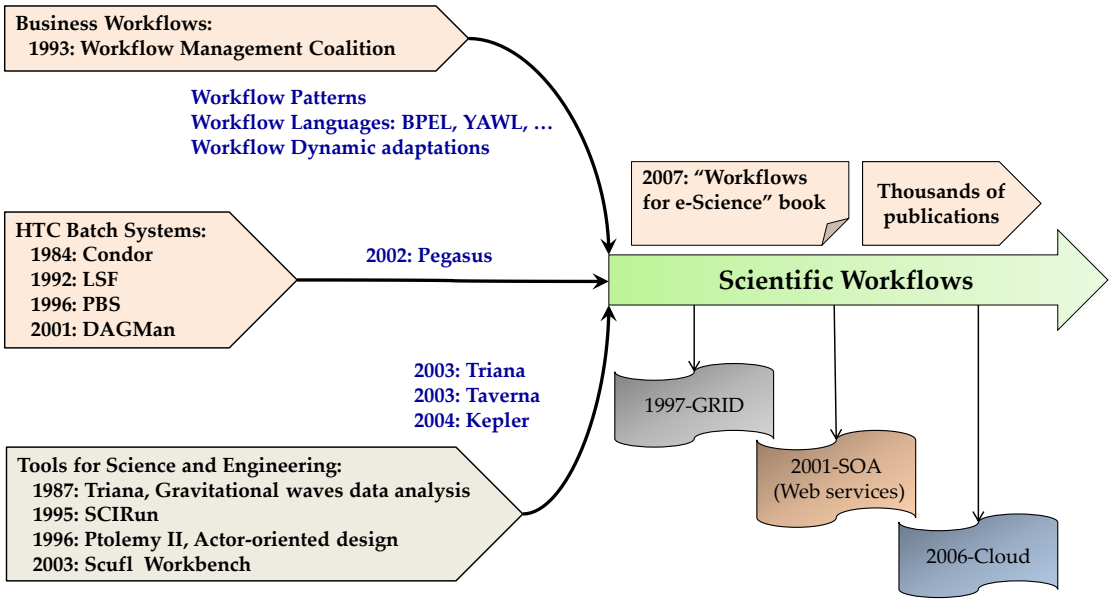


Figure 2.1: A global perspective of the scientific workflow initiatives

Scientific workflows use graphs for specifying dependencies between activities in complex scientific applications.

The evolution of scientific workflows was influenced and inherited the best practices followed by three main research directions.

The first important influence came from previous research on business workflows. The workflow paradigm, early adopted in the business context [Hol95] was also found useful to model scientific applications, and allowing flexible mappings between the application abstractions and the underlying computing infrastructures. The main contributions are related to well known standards [WfM11], workflow patterns [Aal+00b], [AHR11], and workflow specification languages, such as the Business Process Execution Language [BPE06] and the YAWL (Yet Another Workflow Language) language [RH09]. Due to

the needs of companies for constant adaptation for changing the business models some workflow initiatives have also dealt with the support for dynamic workflow adaptations [RD98].

The second influential research direction was related to approaches for application decomposition into multiple components and their submission, as batch jobs, for execution on distributed computing infrastructures. The supporting platforms, called *distributed batch computing systems* or *high throughput (HTC) batch systems*, provided job submission, scheduling and management as well as resource management functionalities. Users submit their jobs to the batch system that decides when and where to run the jobs, allows monitoring their progress and ultimately informs the user upon job completion. Successful batch execution systems, such as LSF (Load Sharing Facility) [Zho92], PBS (Portable Batch System) [HT96] and HTCondor [TTM05], have been used both in the scientific and the business worlds. However, these systems raise complex issues concerning the specification and management of dependencies between the application jobs. For instance, the Directed Acyclic Graph Manager (DAGMan) [Cou+07] is a meta-scheduler for HTCondor that manages dependencies between jobs at a higher level than the HTCondor scheduler. The job dependencies are represented by the arcs of a Directed Acyclic Graph (DAG) whose vertices represent the jobs. According to this graph, DAGMan submits jobs to HTCondor that finds computing nodes for executing each job.

The above batch job processing functionalities are used by Pegasus [Dee+15] that is a workflow management system widely used to map abstract workflows into concrete execution plans specified as DAGMan graphs for executing the workflow activities as jobs on distributed platforms using particular middleware, in particular HTCondor. A detailed discussion of Pegasus and its related work is presented in Section 2.2.

The third important influence that brought contributions to the scientific workflows evolution is related to tool-based problem-solving approaches for developing scientific and engineering applications in specific domains. Such approaches were typically supported by sets of tools integrated into Problem Solving Environments (PSE) [GHR94], providing high-levels of transparency to the end user (a scientist or an engineer), with easy to use interfaces and transparent access to parallel and distributed computing resources. A PSE typically encompasses the entire application development and execution life-cycle, by supporting problem specification using a domain-specific language; selection of the application software components, for computation, control or visualization, and their interconnection using data-flow pipelines; followed by the configuration and activation of experiments by setting up the application parameters and mapping the components into the computing platforms; and including the execution management, possibly with monitoring, visualization and steering.

Some PSE initiatives are still in use up to the present days and even originated tools that are nowadays widely used. For example, the SCIRun PSE [PJ95] is a workbench for visual programming developed by the SCI group at the University of Utah. Originally developed for calculations in computational medicine, SCIRun has been extended to

many other application areas [JPW00].

With the emergence of the concept of scientific workflows in the early years of the 21st century, some PSE have integrated the support for scientific workflows in their functionality and architecture [Wal+00]. As examples, we can indicate the cases of Triana, Ptolemy II and the SCUFL workbench described in the following. Triana has initially been developed for analyzing data related to the detection of gravitational waves [TS98]. With the advent of grid initiatives, Triana was later extended to be used as a scientific workflow system for modeling applications within grid computing and peer-to-peer environments [Tay+03].

Ptolemy II [Bro+08a] is a PSE developed at University of California at Berkeley in the context of the Ptolemy project for simulating the design of concurrent real-time, embedded systems. Ptolemy II supports experimentation with actor-oriented design [Agh86]. Actors are concurrent software components interconnected by ports communicating through messages. A model of computation is expressed as a graph of interconnected actors with semantics determined by a software component called a director, which coordinates the actors execution. Despite the possibility of developing new directors, Ptolemy II offers several directors [Bro+08b] supporting distinct models of computation, such as process networks (PN), discrete-events (DE), synchronous data-flow (SDF), synchronous/reactive (SR), continuous-time (CT) and a *rendez-vous* model.

Built upon Ptolemy II, the Kepler scientific workflow system [Alt+04] inherits the maturity of Ptolemy II, namely the supported models of computation as directors. The Kepler project has been developing extended characteristics for allowing the development of scientific workflows.

The SCUFL workbench [Add+03] is a software graphical tool for designing workflows using the workflow language SCUFL (Simple Conceptual Unified Flow Language). This workflow language was developed as part of the *myGrid* project [GWS03] related to e-Science communities, mainly in the areas of bioinformatics and the mapping of the human genome. The SCUFL workbench is also a workflow enactor to execute workflows accessing Web Services hosted on distributed computing resources. The Taverna workflow system [Oli+06] integrates the SCUFL workbench and uses the SCUFL workflow language for supporting the development of scientific workflows.

Some of the workflow systems identified above have been widely used until today or have been the root of multiple related works to explore specific characteristics of scientific workflows. Therefore, in Section 2.2 we describe in more detail these workflow systems, such as Pegasus, Triana, Taverna and Kepler.

In the meanwhile, from the early years of the 21st century, the emergence of e-Science initiatives [FK03; Hin06; NeS12] claimed for improved software tools and environments towards more productive modeling and experimentation with physical and virtual phenomena in diverse scientific application domains. This has been introducing continuously challenges and new requirements in scientific workflows systems.

On one hand, this involves complex computational processes for simulation, visualization, access to large data sets, and increasing degrees of interaction with the user. On the other hand, it involves the establishment of standards related to Service Oriented Architectures (SOA), and the emergence and availability of large computing resources on grid and cloud infrastructures.

As a curiosity, in January 2016 a Google search for the *"Scientific Workflows"* sentence gave 110.000 results. This reveals the impact of scientific workflows in multiple distinct application domains leading to thousands of scientific publications. As an important indication of such impact in the year of 2007, the book *"Workflows for e-Science: Scientific Workflows for Grids"* [Tay+07] was published. This book represents a milestone by providing a survey of the state of the art and by discussing the future directions for scientific workflows. The book also includes descriptions of the major existing scientific workflow systems in particular Pegasus, Triana, Taverna and Kepler.

2.2 Widely Used Scientific Workflow Systems

In spite of intensive research efforts on scientific workflows and many early achievements, unfortunately, there are only a few general purpose and flexible workflow systems that can be effectively used by end users in different science domains [BH08; CG08; Gil+07; McP+09; Tal13; YB05]. In spite of the large number of available scientific workflow systems and tools, there is no consensus about what a general purpose scientific workflow system is. In fact, some systems have been developed to support research on specific science domains. For instance, Galaxy [Goe+10] and BioExtract [LGD15] are systems closely related to bioinformatics. Despite the multiple contributions and achievements from many other systems, in the following we only discuss scientific workflow systems that are used in various science domains and that are widely cited in publications related to scientific workflows: Pegasus [Dee+05], Triana [Chu+06], Taverna [Oli+06] and Kepler [Lud+06].

2.2.1 Pegasus

Pegasus [Dee+05] is a framework for mapping scientific workflows onto distributed computational resources based on grid or cloud systems [Dee+16].

In Pegasus an application is modeled by an abstract workflow specified in a high-level language as a directed acyclic graph (DAG). At abstract level a workflow specification is described using a proprietary language named DAX (as an acronym for Directed Acyclic Graph in XML), which allows representing a DAG in XML. The workflow activities, named tasks, are represented by the graph nodes and the data dependencies between tasks are represented by the graph arcs. Pegasus transforms the abstract workflow to a concrete workflow by mapping the workflow tasks to the existing computational resources where the workflow application is executed. This concrete workflow is defined by

a set of script files specified using the meta-scheduler DAGMan, [Cou+07] that enforces the dependencies between tasks and submits the workflow tasks to be executed by the HTCondor [TTM05] batch system. When a workflow task fails, DAGMan can retry its execution. If the task continues to fail, DAGMan generates a rescue workflow that can be resubmitted later.

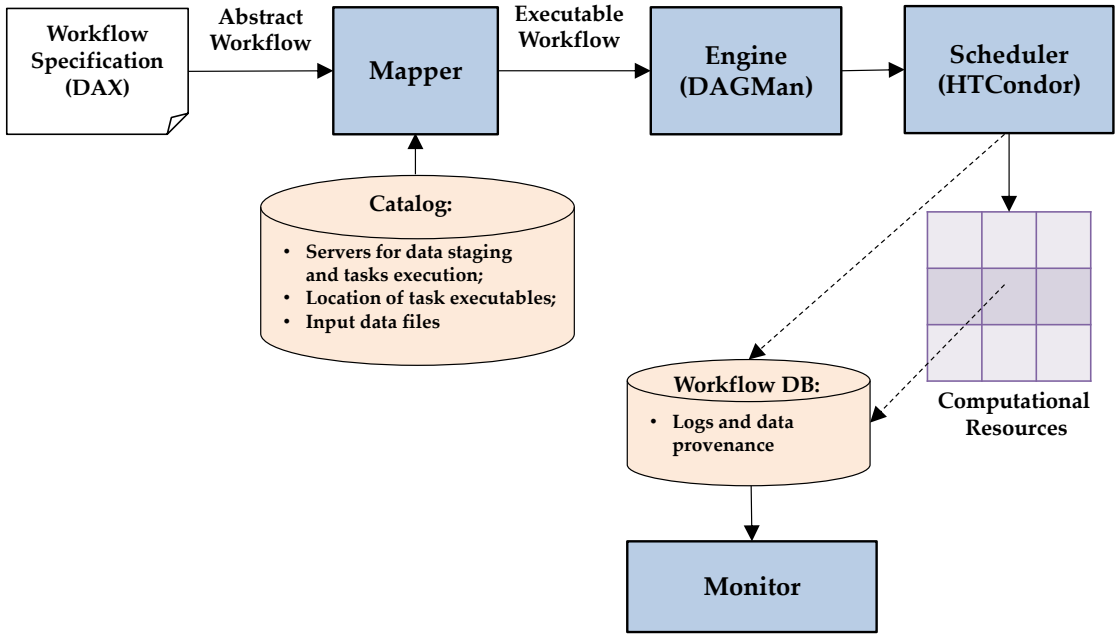


Figure 2.2: The Pegasus system components, [Dee+16]

As illustrated in Figure 2.2 the Pegasus system is composed of the following four components:

1. *Mapper:* Based on the user DAX workflow specification, Mapper finds the adequate computational resources required for workflow execution in a catalog, such as the software components mapped to tasks and data repositories and generates an executable workflow. For optimization purposes the Mapper can restructure the workflow by adding new tasks for data management including the generation of provenance information;

2. *Engine (DAGMan):* According to the workflow task dependencies of the executable workflow, DAGMan determines when tasks are ready to be executed and then submits them to the HTCondor queue for execution. The engine also checks the workflow failures and in case of a task failure retries its execution;

3. *Scheduler (HTCondor Scheduler):* The *Scheduler* manages a queue of tasks and their execution on the available computational resources;

4. *Workflow monitor:* The *Monitor* maintains a database with runtime provenance and performance information and notifies the users about task failures. The *Monitor* provides a Web interface for allowing users to monitor their workflows.

The specification of abstract workflows can be described using tools according to the multiple projects that have been using Pegasus. For example, the workflow system Wings (Workflow Instance Generation and Specialization) [Gil+11] automates the creation of large-scale workflows for simulations to construct seismic maps for the Southern California Earthquake Center. The Wings system assists users to create workflows by using artificial intelligent planning and semantic reasoning. The workflows generated by Wings have associated directed acyclic graphs (in DAX) for submission and execution using Pegasus.

However, Pegasus is strongly dependent on the High Performance Computing (HPC) cluster model and the HTCondor scheduler. On one hand, for exchanging data between tasks or even the task data input/output, the Pegasus approach depends on HPC clusters shared file systems. On the other hand, if the HTCondor scheduler is not installed on the computational infrastructure it is not possible to use DAGMan and consequently the Pegasus workflow system. To solve this issue a new workflow engine, DAGwoman [TS12], has been proposed to run DAGMan workflows on clusters and institutional computing resources using other scheduling systems, for instance Sun/Oracle Grid Engine (SGE/OGE).

2.2.2 Triana

Triana [Chu+06] is a workflow system, initially integrated as a problem-solving environment, most used for signal processing. Triana has a friendly Graphical User Interface (GUI) for designing workflows by drag-and-drop of functional tools onto a workspace and connect them for composing applications. Triana also has an embedded subsystem for executing the workflow. As shown in Figure 2.3, the Triana user can design the workflow by simply drag-and-drop of the available Triana tools, connect them by data-flow oriented links, and run the workflow application.

A Triana tool is a software component, called *unit* that implements a well-known interface, which allows the development of new tools to be integrated in Triana for further use on workflow design. *Units* can be grouped to create aggregate tools, called *group units*, for simplifying the design and graph visualization of large workflows.

Behind the graphical user interface for designing the workflow, Triana uses its own specification language based on XML.

Triana allows interaction with remote services through two interfaces, called Grid Application Prototype (GAP) and Grid Application Toolkit (GAT). These interfaces enable Triana workflows to interact with peer-to-peer systems, Web Services or grid tools, such as the Globus Resource Allocation Manager (GRAM) and the Grid File Transfer Protocol (GridFTP). At run time the GAP/GAT interfaces support the workflow refinement by automatically mapping the workflow tasks onto the available distributed resources [Shi07a]. Furthermore, Triana users that are running the graphical interface can log in into a remote Triana Controlling Service (TCS) for running Triana workflows remotely.

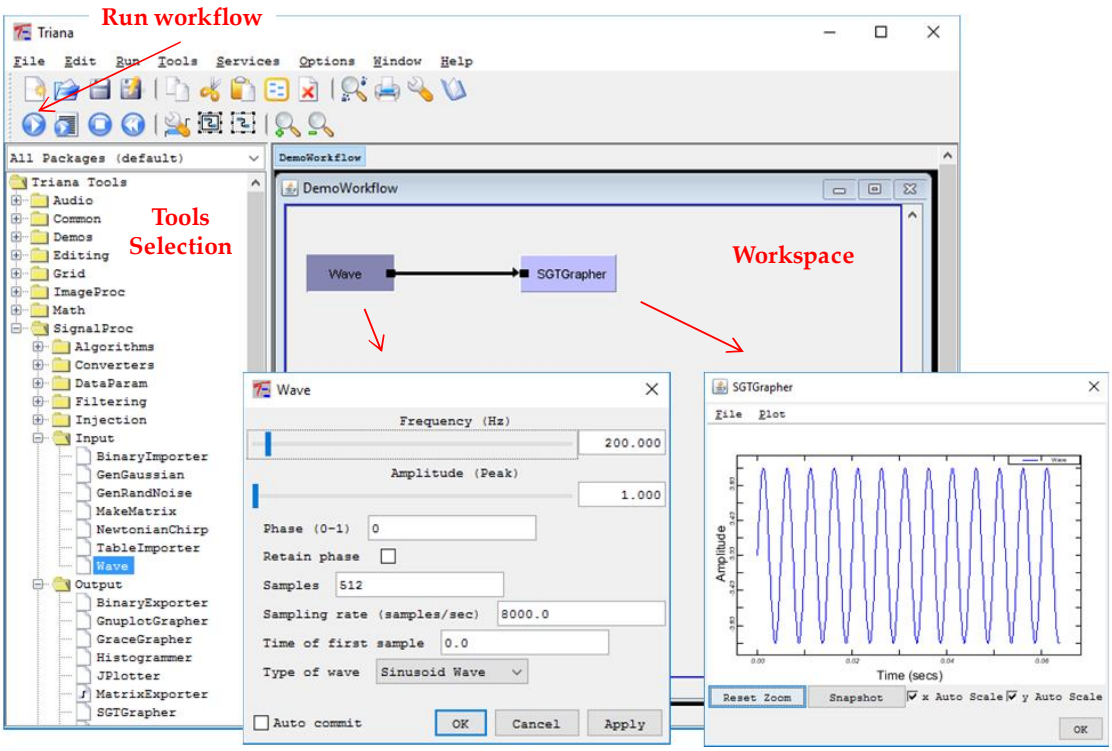


Figure 2.3: The execution of a basic workflow using the Triana user interface, [Chu+06]

Despite the above interactions with remote services, Triana is based on a centralized execution engine even when the workflow execution is performed remotely using TCS.

The links between Triana *units* are typically based on data-flow. However, control-flow links can be supported easily by using special messages between *units*.

Triana has some limitations, for instance, it only supports Web Services synchronous call. This penalizes the performance because it does not support two simultaneously parallel *units* to invoke Web Services [AGC09]. Other limitation is related to supporting complex data types when composing workflows using Web Services. In fact, Triana has special *units*, *WSTypeGen* and *WSTypeViewer* to generate and view complex data types that are useful for testing simple invocations of Web Services. However, if two Web Service invocations use slightly different complex data types, the only way to connect them in a workflow is to develop a new *unit* for ensuring data compatibility.

2.2.3 Taverna

The Taverna [Oli+06] workbench was developed in the context of the *myGrid* project for supporting *in silico* experiments in life sciences and for providing a user-friendly graphical interface allowing scientists to access underlying Web Services.

Taverna relies on an approach based on combining Web Services into workflows. This approach assumes that users think in terms of data processing dependencies by connecting services together independently of the concepts and details of service-oriented

architectures.

Taverna uses the Simple Conceptual Unified Flow Language (SCUFL) for workflow specification. Taverna enables users to design and execute workflows using the embedded workflow enactment engine. A *processor*, the basic execution unit in Taverna, can be viewed as a function of a set of input data to a set of output data, represented as ports on the *processor*. Ports can be connected by data links to support data-flow or by coordination links to synchronize the execution of components.

As shown in Figure 2.4 the Taverna workbench has three areas: The *Service Panel*, used to manage services, allows to choose or import Web Services; The *Workflow explorer* is used to specify the workflow without direct interaction with the SCUFL language; and the *Workflow diagram* is a visualization canvas used to represent and interact with the workflow diagram.

The workflow example represented in Figure 2.4 allows getting the weather forecast for a zip code.

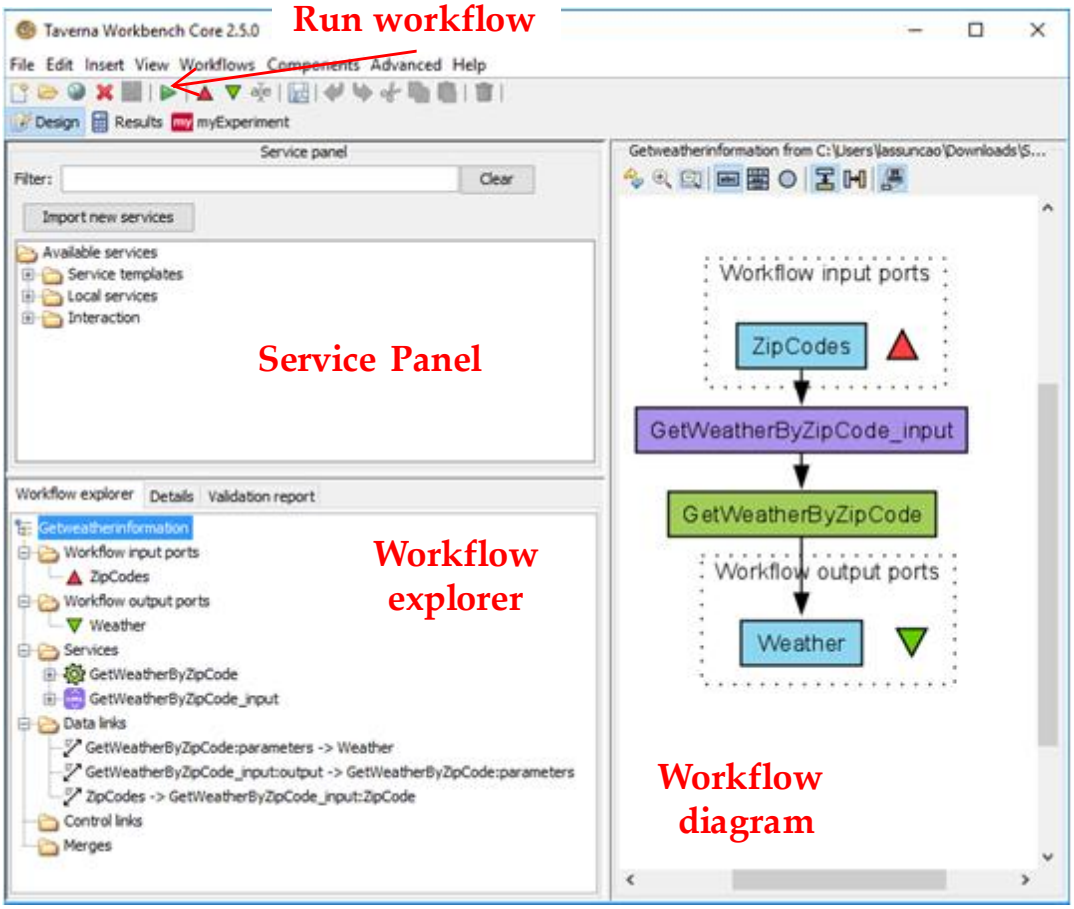


Figure 2.4: The design and execution of a basic workflow using the Taverna workbench, [Wol+13]

Taverna was developed for satisfying the needs of bioinformatics scientists that mainly

need to build scientific workflows from a diversity of remote Web Services. Therefore, a significant characteristic of the Taverna system is the organization of Web Services into a reusable collection of components. Taverna only assumes the XML data format and then some of these components encapsulate data types that are resulting from the Web Service specification based on the standard Web Service Definition Language (WSDL).

2.2.4 Kepler

The first experiments made with scientific workflows in the context of this dissertation [AGC09] were based on Kepler. This decision was due to the following Kepler characteristics:

- Ease of download, including the Java source code;
- Flexibility of the installation and configuration on Windows and Linux operating systems without dependencies on any middleware except a Java Virtual Machine;
- An intuitive and user-friendly Graphical User Interface (GUI) allowing design and execution of workflows;
- A very comprehensive documentation [Kep14], inherited from the Ptolemy project including a detailed description of the user interface.

Kepler is an open-source Java-based environment for building scientific workflows [Kep13; Lud+06]. The core of Kepler is based on the Ptolemy II environment [BL08] built for modeling, designing and simulating concurrent systems.

Kepler provides a very intuitive Graphical User Interface (GUI), Figure 2.5, inherited from Ptolemy II where the workflow components (*actors*, *links* and *directors*) are dragged and dropped onto a canvas where they can be interconnected, customized and executed.

Models in Ptolemy II and Kepler are based on the composition of *actors* [BL05] that are abstract boxes to encapsulate parameterized actions used to wrap different types of software components, for example, local application jobs or remote Web Services. The role of *actors* is receiving input tokens from input ports, processing these tokens and producing output tokens to output ports. *Actors* are connected using *links* between output and input ports, forming workflow graphs.

Kepler decouples the workflow model from the execution engine by assigning one model of computation enforced by a *director* to each workflow.

The interaction between *actors* is orchestrated by a *director* that is responsible for implementing the communication semantics among ports and the control or data flow among *actors*. Like in cinema the *director* directs the action looking at how the *actors* are connected, moving data from one *actor* to others, and firing the *actors* when they should execute.

One specific *director* implements a Model of Computation (MoC) that defines the scheduling and execution semantics (orchestration semantics) for workflows.

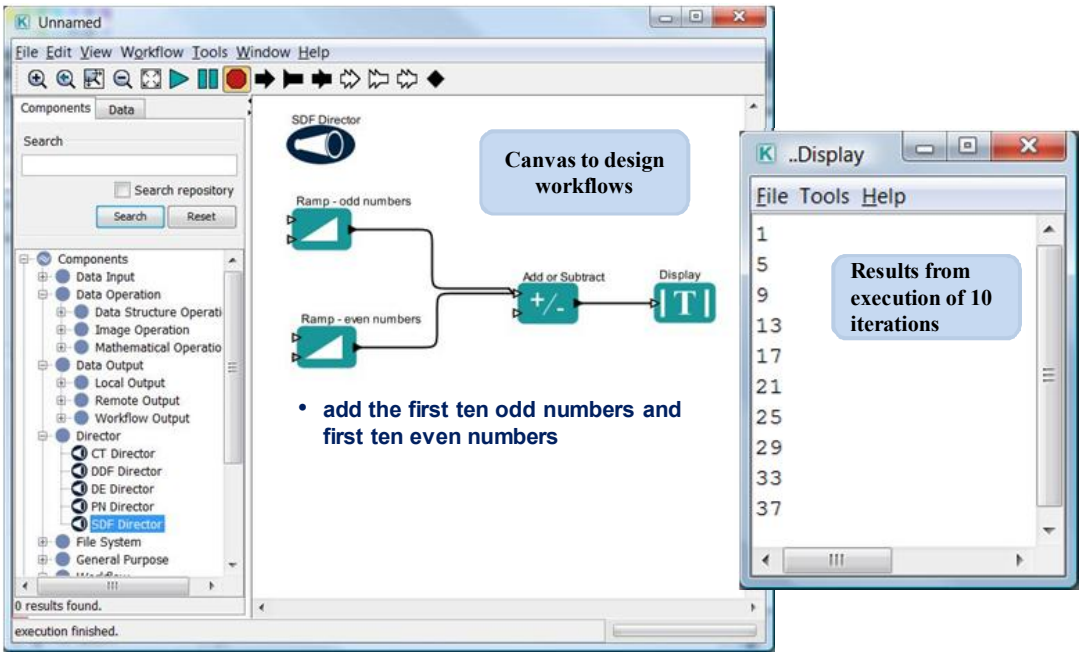


Figure 2.5: The execution of a basic workflow using the Kepler user interface, [Kep14]

Ptolemy II defines eleven models of computation (MoC) implemented as *directors* that are described in detail in [BL08] (Vol. 3: Ptolemy II Domains). A subset of *directors* is available in Kepler. The Continuous Time (CT) and Discrete Events (DE) directors are used in workflows with time dependencies and the Synchronous Data Flow (SDF), Process Networks (PN) and Dynamic Data Flow (DDF) *directors* that are used in workflows without time dependencies.

Despite the existing guidance from the Kepler User's Manual [Kep14], the questions how to choose *directors* and the different ways how they work are quite subtle. *Directors* establish data-flow oriented structures for enabling the execution of the *actors* with data tokens available in all input ports.

Kepler allows remote data access by file movements between locations using specific *actors* that wrap middleware functions, for instance using protocols like GridFTP[1] or shell commands like SCP[2].

Kepler does not have any fault handling mechanism at *actor* level. However, at the workflow level when an exception occurs the workflow execution stops and an exception window is displayed where a user can analyze the stack trace and can decide if it is possible to dismiss the exception and resume the execution.

According to the *director* properties, workflows can be configured for executing long-running workflows with multiple iterations. During the workflow execution a user can

[1] GridFTP is an extension of the standard File Transfer Protocol (FTP) used in grid environments. It is defined as part of the Globus toolkit (http://toolkit.globus.org).

[2] The Secure Copy Protocol (SCP) is a network protocol that allows users to copy files securely between computers in a network.

graphically observe the workflow execution (Animate at Runtime in the Tools menu). However, there is no way to dynamically change the structure and behavior of the long-running workflows.

Kepler workflows are represented and easily exchanged in XML using the Modeling Markup Language (MoML) inherited from Ptolemy II.

Despite some Kepler extensions to spawn distributed executions [Plo+13], by default, *actors* in Kepler run as local Java threads inside a monolithic operating system process.

To summarize, Kepler has the following strong characteristics:

- An intuitive and user-friendly graphical interface;
- A large user community in different science domains;
- A component library with built-in *actors* (data input/output from files, databases etc.) including specific *actors* for accessing Web Services or distributed resources on clusters and grids;
- Different models of computation supported by different *directors*;
- The possibility of developing new *actors* in the Java language and using them in specific workflows.

However, we also found several important limitations:

- The complexity of the data type model and the *actor* model for developing new *actors*. In fact, to develop a new *actor* the developer needs to understand the interaction model between *directors* and *actors* as well as the rules for passing tokens between *actors*. In some workflow scenarios the user needs to deal with low-level data types, such as *RecordToken*, *XMLToken* etc.;
- The *actor* for invoking Web Services is not flexible to support tokens based on the complex data types defined by the Web Service contract specified using the Web Service Definition Language (WSDL);
- Despite the flexibility for using distinct computation models by choosing distinct *directors*, some built-in *actors* do not work properly with all *directors*. For example the *PN director* does not manage iterations, possibly leading to non-determinism and undefined termination of the execution. In *composite actors* with workflow hierarchies, if two *actors* have computation threads, it is ambiguous which *actor* should be allowed to perform the computation [God+09];
- The workflow execution engine and *actors* run as local Java threads inside a monolithic operating system process. This characteristic inherited from Ptolemy II represents a limitation for executing large workflows with many activities. Furthermore this characteristic poses difficulties to the distributed execution of the workflow activities.

2.3 Replicability and Reproducibility of Scientific Workflows

In order to support *in silico* experimentation in some science domains it should be possible to repeat experiments for validating results as well as for disseminating the experiment and some results between scientists. [Dru09] presents the concept of replicability as the possibility of recreating an experiment, using the same tools, the same data and equivalent computing resources, and the concept of reproducibility as the possibility of repeating an experiment where we always get the same output for the same input, independently of the tools and computing nodes used. Aiming at similar goals related to managing the evolution of workflow execution, the concept of provenance has been addressed as an important characteristic for scientific workflows [DF08; Mat+13; Mil+08].

Other initiatives use artificial intelligence algorithms and semantic Web languages, such as the World Wide Web Consortium (W3C) standards, Ontology Web Language (OWL), Resource Description Framework (RDF), Simple Protocol and RDF Query Language (SPARQL) to describe a workflow execution, including the input data, the computation steps and data results [Gar+12; GGC14]. These workflow descriptions are later used to assist in future workflow compositions and their validation [Gil+11; GR16].

However, as discussed in [SP+16; Zha+12] the existing workflow systems still have limitations on the adequate levels for supporting the replicability and reproducibility properties.

We also consider that the difficulties related to the availability of workflow systems and tools for downloading, installation and configuration also introduce limitations for supporting the replicability and reproducibility properties. Nowadays, it is easy to create and configure virtual machines (VM) with an entire working environment, including the workflow system and its dependencies on other software components or even data scripts and logs related to particular experiments. Furthermore, a snapshot of a stable working environment can be saved on VM images that can be made publicly available. These VM images facilitate the replicability of the working environment by allowing to instantiate new virtual machines or even to reproduce scientific experiments. The availability of cloud providers and the flexibility to host a large number of distinct virtual machines ease the creation of replicable and reproducible scenarios. For instance, the Amazon cloud infrastructure provides a Linux virtual machine image that includes the Pegasus workflow system, the HTCondor worker daemons, and other packages required to compile and run the tasks of the selected workflows, including the application binaries [Juv+10].

Despite the advantages and opportunities of using virtual machines on clouds to facilitate the replicability and reproducibility of scientific workflows there are still many challenges to be met [How12].

2.4 Dimensions of the Scientific Workflow Characteristics

In the last decade an extensive research work has encompassed multiple dimensions related to scientific workflows. However, the scope and level of those dimensions are not easily characterized and some works present overlapping distinct dimensions. Scientists of distinct areas of science tend to classify the workflow characteristics according to their interests. Most of the existing surveys on scientific workflows, typically, are focused on specific characteristics or on a description of the basic concepts and the corresponding implementation in the existing workflow systems. Since the year of 2007, when the book *"Workflows for e-Science: Scientific Workflows for Grids"* [Tay+07] was published, multiple publications have surveyed the characteristics of scientific workflows and discussed how those characteristics are more or less fulfilled by the major existing scientific workflow systems.

A concise survey of the technologies for supporting workflows in the business and scientific domains is presented in [BH08]. This work concludes that workflow tools need to be developed according to the science domain instead of being built by computer scientists using highly specific and hardly used characteristics. As an example, the authors argue that biologists are uncomfortable to think in terms of service abstractions currently available in the existing workflow tools.

Illustrated with workflows in different science domains, including bioinformatics, astronomy, and weather and ocean modeling, [RG08] discusses the requirements and the constraints to manage workflows that are using distributed resources. In [RP10] the authors present a classification model of the workflow characteristics based on a set of workflow examples from multiple science domains. The classification model defines the following workflow characteristics: i) The size of the workflow, considering some properties, such as the total number of workflow activities, the maximum number of parallel branches and the number of activities in the longest branch of the workflow; ii) The dominant structural patterns in the workflow, such as sequence, parallel split, and synchronization, according to [Aal+00a; Aal+00b]; iii) Data patterns, including input, intermediate and output data as well as the data types and the transfer time associated to the workflow data-flow; iv) The usage scenarios, such as interactive workflows where a user can inspect intermediate data and make changes during the workflow execution.

In [YB05] an exhaustive taxonomy is presented, classifying the approaches for building and executing workflows on grids and also a survey of the existing grid workflow systems. Despite the context of grid workflows many of the characteristics also apply to scientific workflows. Illustrating well the complexity for enumerating all characteristics of the workflow systems and tools, the classification taxonomy tree has 84 nodes for defining, among others, characteristics related to the workflow design, for instance, the workflow structure for supporting DAG and Non-DAG with loops, centralized and decentralized task scheduling, fault tolerance at task level and workflow level, and characteristics related to data movements. However, some characteristics are not well identified.

For instance, characteristics related to the support of structural and behavioral dynamic reconfiguration for long-running workflows are not considered. The work only considers dynamic characteristics for retrieving information related to the selection of the adequate resources for task scheduling and execution.

In a specific science domain, [Wan+10] presents a survey of scientific workflows for astronomy. The authors argue that scientific workflows are significantly different from the business workflows, pointing out that scientific workflows use more data-flow patterns while business workflows use more control-flow and event patterns. The work also discusses various drawbacks of the existing workflow systems related to three aspects: i) Lack of flexibility and complexity for using software tools including some graphical interfaces; ii) Difficulties to understand the architecture and to extend the functionality of the workflow systems. As an example, the authors argue that only experts can easily develop *actors* in the Kepler workflow system; iii) Installing and configuring software platforms with complex dependencies is not easy for astronomers. Thus deploying some workflow systems is a hard job for astronomers.

To achieve workflow interoperability, [EHT10] discusses characteristics of the three dimensions that must be considered, such as the workflow execution environment, the model of computation, and the workflow specification language. After discussing distinct approaches for each dimension the work points out the existence of some trade-offs. For example, a complete decoupling between the execution environment and the workflow language improves the interoperability, but also makes the workflow design more tedious and error prone. A trade-off also exists between the execution environment and the model of computation. For example, in some models of computation the workflow activities are tightly coupled to a particular workflow execution environment and it is not easy to reuse those activities in a different workflow execution environment.

In [Tal13] is discussed how basic concepts of scientific workflows are supported in some existing workflow systems and tools, such as Pegasus, Triana, Kepler, Taverna and some grid related workflow systems, such as Askalon [WPF05] and the GridWorkflow Execution Service [Hoh06]. The work also discusses some open research issues in the area of scientific workflows. For example, the author argues that a pure data-flow concept is very hard to implement using a decentralized control on parallel and distributed systems, mainly to ensure fault tolerance. Therefore the author claims for adequate abstractions for data representation and concurrent processing on large-scale computing infrastructures available today for running scientific applications. The work also defines a set of research issues towards the design of the next-generation scientific workflow systems. These issues are related to: i) High-level and complex abstract structures to be included in workflow programming tools; ii) The support for distributed workflow execution on service-oriented and cloud infrastructures; iii) Techniques for dynamically adapting the execution of workflows; iv) Fault tolerance and recovery in scientific workflows; v) Techniques for managing, visualizing and mining provenance data to face the Big Data challenge.

Many works are related to scheduling techniques to map the workflow execution to multiple computational nodes. The scheduling of N tasks on M processors is a well studied NP-complete optimization problem [BA04; Ull75]. The issue of launching workflows with a large number of activities to a possibly scarce number of computational nodes with possible constraints has been the subject of extensive research [Bru07; KA99], also in the context of workflows [Che+15; DK07; IT07];

More specific surveys, such as related to failures and exception handling, and workflow steering have been published in works described in the following sections.

However, on one hand, the multiple stakeholders involved have difficulties in presenting a consensual characterization of the multiple dimensions of the scientific workflows. On the other hand, it is unpractical to attempt an exhaustive discussion of all related work due to the huge number of existing publications related to scientific workflows. Nevertheless, in the following section we discuss the main dimensions and the related work that influenced the development of the AWARD model and its implementation.

2.4.1 Generic Characteristics for Scientific Workflows

In this section, we first discuss characteristics concerning the expressiveness of the workflow specification and then we discuss characteristics related to the workflow computation models.

⊸ Expressiveness of the workflow specification

The expressiveness of the workflow specification has been discussed in many formal and theoretical works including: Workflow specification languages [Aal+04; AMA06; FQH05]; Structural workflow patterns [Aal+00a; Aal+00b; KHA03], workflow data and resource patterns [RHE05] and more recently in scientific workflows context [Mig+11] as well as techniques for verifying the correctness of workflows, for instance detection of deadlock and lack of synchronization [AH00; Sor+07]; The Petri net formalism [BC92; Mur89] for modeling workflows supporting both control-flow and data-flow [AH00; Gub+06; HA07; TC09; TCBR11]; Models to support autonomic and decentralized workflow execution [HPA05; LP05; NPP05; NPP06].

✧ Workflow languages

In the context of business workflows, including commercial systems, multiple specification languages have been proposed. In [AH05] thirteen commercial workflow systems are described, where each one has its own specification language and also ten more languages proposed by the academic community. This plethora of workflow languages, some of them being proprietary, led to initiatives to define independent languages.

The design and implementation of a workflow system, which uses the Yet Another Workflow Language (YAWL) is presented in [Aal+04]. The YAWL language supports most of the workflow patterns [Aal+00b] and has a formal semantics defined as a transition system based on Petri nets. The YAWL expressiveness and its formal semantics aimed to

define a standard intermediate language for supporting language interoperability. However, to the best of our knowledge the YAWL language implementation relies on a centralized workflow execution engine. Although YAWL has been extended to support data exchange using XML, its initial focus was on control-flow for supporting dependencies between workflow activities.

The Abstract Grid Workflow Language (AGWL) used for specifying grid workflow applications at a high level of abstraction is presented in [FQH05]. AGWL is a XML-based language decoupled from low-level details of the underlying grid infrastructure. However, AGWL is compromised with the ASKALON workflow runtime environment [WPF05].

Despite the efforts within the Workflow Management Coalition (WfMC) [WfM11] for defining workflow standards to be used in the business domain, most earlier commercial workflow systems relied on proprietary workflow languages. However, in the first years of the 21st century, the establishment of standards for Web Services led to the emergence of the Business Process Execution Language for Web Services (BPEL4WS) (or simply Business Process Execution Language (BPEL)) that in 2004 was considered a standard within the Organization for the Advancement of Structured Information Standards (OASIS) [OAS15b] and has been adopted by the majority of the available commercial workflow systems. The BPEL syntax relies on the XML language and the workflow execution engines, typically, run inside centralized business application servers.

Some scientific workflow initiatives have considered BPEL as a candidate language for specifying scientific workflows. [AMA06] evaluates how BPEL meets the requirements of scientific workflows and concludes that it is a good candidate for orchestrating Web Services but has limitations for reusing primitive workflow activities and for supporting user interactions. [Slo07] discusses how to adapt BPEL to scientific workflows and concludes that BPEL is a viable choice for a grid workflow but BPEL workflow engines need additional capabilities to address some grid requirements. As BPEL was designed for using Web Services the authors found difficulties to integrate legacy scientific software.

Despite the efforts for trying to define a standard language for specifying scientific workflows, unfortunately, this objective has not yet been reached. In fact, most of the existing scientific workflow systems, such as Pegasus, Triana, Taverna and Kepler, are using their own specific languages with their specific XML dialects.

✧ **User interfaces**

The user interface supporting the specification of workflows is an important characteristic. For example, the Triana, Kepler and Taverna workflow systems have powerful and user-friendly graphical interfaces, which have contributed to their wide adoption.

Typically the user environment associated with a workflow system has a graphical user interface with a canvas area where the workflow developer designs the workflow graph without having to deal with low-level scripts or XML dialects as a specific workflow language. It is the tool that automatically generates the workflow specification file according to the used workflow language.

However, most of the existing tools supporting the workflow graphical design have limited support for editing and displaying parts of large workflow graphs. For example, [RG08] describes two workflows (Avian Flu and PanSTARRS) that have over a thousand of activities. Although this dissertation does not address the support for designing workflow graphs, the design of workflow graphs with hundreds or thousands of activities is still an important open issue.

✧ **Workflow patterns**

Several works from the business workflow domain tried to define workflow patterns for systematically identifying the workflow requirements and functionality. For example, [Aal+00a; Aal+00b] have identified more than forty patterns from the basic to advanced workflow patterns aiming to establish a comparison among the commercially available workflow management systems.

[Rus+05] identifies workflow data and resource patterns describing how data and resources are represented and used in workflows.

[AH00; Sor+07] discuss techniques for verifying the structural correctness of workflows, for instance, deadlocks detection and lack of synchronization.

In the scientific workflow context, [Mig+11] presents a pattern-based evaluation of the well-known scientific workflow systems, such as Kepler, Taverna and Triana, and compares them with other business workflow systems. The authors observed that some patterns are not directly supported in scientific workflow systems. These patterns, classified as routing patterns, are related to the flexibility for combining and routing the input tokens before the task execution. This work also claims for the need of integrating human agents into scientific workflows.

⊸ **Computation models**

Different approaches have been proposed for modeling and expressing the computation models governing the workflow execution.

✧ **Process Networks**

The Process Networks(PN) model of computation [Kah74] is a powerful computation model that implicitly supports parallelism between a group of processes that communicate through unbounded FIFO channels by reading and writing atomic data elements called tokens.

The PN model of computation assumes that the write operation is non-blocking and always succeeds, while the read operation is blocking, that is reading tokens from an empty channel blocks the process until at least a token is available. The PN model does not allow processes to check input channels to test for the existence of tokens without consuming them.

The PN model leads to programs with a determinate behavior, that is given a trace of input tokens a process computation is deterministic, always producing the same trace of output tokens, thus the execution order of processes does not affect the output trace of tokens.

As pointed out by Parks [Par95], due to the equivalence of a PN process network to a set of interconnected Turing machines, both termination and boundness are undecidable in finite time. Namely the question whether a PN network is strictly bounded is undecidable, that is, to determine in finite time if any execution of a PN network requires bounded memory. This introduces important practical issues concerning the implementation of PN networks.

The first approach to address the boundness of PN networks is due to [Par95], where execution starts with processes connected by bounded FIFO channels, of predefined initial sizes, and the execution proceeds in a data-driven way until a deadlock occurs, that is all processes are blocked on full or empty channels. In this case, if at least one process blocks on a full channel, the deadlock is said artificial as it would not have occurred in a PN network with unbounded channels; otherwise, the deadlock indicates termination of execution (with all processes blocked on input channels). Artificial deadlocks are resolved at runtime by increasing the capacity of one of the full FIFO channels.

Since Parks' original proposal, several approaches have been developed based on the same idea [BH01; GB03]. However, most of them are restricted to single-processor or multiprocessor architectures.

Although the PN networks boundness problem has been studied also for distributed-memory PN frameworks [AZE07; Ama+03; OE06; PR03], and there are proposals that try facing the challenges posed by the lack of global state and the difficulties of distributed deadlock detection and resolution to the best of our knowledge there is a lack of effective working implementations of solutions to this issue, in real PN distributed frameworks.

✧ **Petri nets**

Petri nets, introduced in 1962 by Carl Adam Petri, are a graphical and mathematical formalism used for modeling the structure and behavior of large set of system types [BC92; Mur89], namely for modeling the semantics of concurrent and distributed systems. The flexibility for representing such systems as Petri nets and the recognized capability for analyzing and proving properties on the behavior of the system using formal techniques, have led Petri nets to be the most used model for analyzing the workflow behavior. Therefore multiple workflow systems, both in business and scientific domain, rely on the semantics of Petri nets for describing the structure and behavior of workflow patterns. For example, [AH00; KHA03] describe basic workflow control-flow patterns using Petri nets. [Gub+06] presents a model and a tool for composing scientific workflows based on the Petri nets formalism. [HA07] discusses how Petri nets can be applied for managing workflows towards the choreography, orchestration, and execution of e-Science applications.

[GRC08] also use Petri nets for representing scientific workflows and for demonstrating how patterns and operators can be used to adapt the structure and behavior of a Triana workflow.

Some variants of Petri nets were also used for modeling particular characteristics of scientific workflows. For example, based on Reference nets, as a particular class of

Petri nets, [TC09] proposes a scientific workflow engine, called DVega, with special emphasis on the flexibility for supporting hierarchical workflows capable of handling and propagating exceptions in the hierarchy. The exception handler can dynamically adapt the workflow by replacing a subworkflow in the hierarchy with an alternative subworkflow without affecting the rest of the workflow structure. The alternative subworkflow is chosen from a list of previously defined candidates.

✧ **DAG and non-DAG**

Many cases of scientific workflows can be described as a directed acyclic graph (DAG), where the nodes represent computational tasks and the edges represent dependencies between them. For instance, in [Gar+12] a catalogue of scientific workflow motifs is proposed as a result of an empirical analysis performed over 177 DAG workflows. Therefore most of the workflow management systems, including Pegasus, Triana, Taverna and Kepler have support for executing DAG workflows.

However, some workflow scenarios [AGC09] require a structure involving feedback loops mainly when the workflow is to be executed with multiple iterations. Iterations and feedback loops are not supported in many existing workflow systems, for instance, in Pegasus.

Triana supports loops and execution branching, handled by specific *units* with not easy to use semantics.

Kepler supports iterations and a particular case of feedback loops using the built-in *SampleDelay actor*, which outputs a set of tokens that are used as initial input values when a workflow starts up and for each iteration it simply carries its input port value to its output port at the current iteration.

As discussed in Section 2.2.4, Kepler is a flexible workflow system with support for multiple models of computation inherited from Ptolemy II. However, the most useful models concerning scientific workflows are the Synchronous Data Flow (SDF) and the Process Networks (PN), [AEA08]. Most of the *composite actors* built with Kepler to be used in scientific workflows have been based on the PN model. However, nesting a PN *director* inside a subworkflow with a SDF *director* would be invalid in most cases [God+07]. The Kepler PN model is based on Kahn Process Networks (PN) [Kah74]. As an extension of the Kepler system, COMAD [McP+09] extends the Kepler process network (PN) data-flow model to facilitate the management of scientific data within Kepler scientific workflows.

✧ **Pipelines**

A pipeline is a particular structure of a workflow with a sequence of tasks that continuously process a stream of items, where each item is pumped from one task to its successor as soon as possible. Typically, items are produced by partitioning the input data into chunks with the same size. The pipeline execution model is used for concurrent processing on many different types of middleware platforms. However, for processing multiple items in parallel, the workflow execution engine needs to support an efficient execution scheduler. [Ben+13] presents a survey of pipelined workflow scheduling techniques concluding that, despite the significant body of literature, there is a need for more

work, both theoretical and practical, in order to support realistic application scenarios.

As an example, [Thr+10] discusses how to execute a bioinformatics pipeline workflow used for sequencing and extracting representatives of the mRNA[3] of a cell.

Some workflow systems and tools have given special attention to pipeline workflows. For instance, Taverna 2 [Mis+10], a new version of the Taverna workflow system, has an improved internal architecture in order to efficiently support streaming pipelining where the workflow activities can process in parallel sequences of unbounded length of input data sets that are continuously produced by a predecessor activity.

✧ **Tuple spaces**

The concept of tuple space was introduced in the Linda model [CG89] for separating the concurrent coordination logic from the application logic. A tuple space is a memory abstraction for storing and retrieving data as tuples (an ordered list of typed fields) that is shared among all interacting participants. The participants interact with the tuple space through a simple interface: i) The write operation for storing tuples; ii) The read operation for reading tuples without removing the tuples (non-destructive read); iii) The take operation for retrieving and atomically removing tuples from the tuple space.

Tuples are read and retrieved by associative addressing using tuple templates. A tuple template contains values of any subset of fields of the tuple to be retrieved and a wildcard, for instance, the character "?" for representing any value of a field.

The implicit parallel coordination logic of tuple spaces is based on the following mechanisms: i) Operations for reading or taking tuples block the execution of the invoking thread until a matching tuple can be found; ii) Any application participant can register a tuple template in order to be sent an asynchronously notification when tuples matching the template are written into the tuple space.

The tuple space concept was introduced in 1989, and in the last decades a large number of implementations of the tuple space model have been developed and available on middleware platforms, such as IBM TSpaces [Leh+01], GigaSpaces [Gig12], or JavaSpaces [Ora12]. Despite the increasing difficulty to ensure the tuple space semantics, namely concerning consistency, in recent years multiple distributed versions of the tuple space concept have also been implemented. Comet [LP05], is a scalable peer-to-peer coordination space that provides a global shared tuple space accessed independently of the physical location of the tuples and the peers. Tupleware [Atk08] is a distributed tuple space that uses a decentralized approach and intelligent tuple search and retrieval to be used as a cluster middleware for supporting computational intensive scientific and numerical applications.

The simplicity of using tuple spaces for coordinating concurrent, parallel or distributed components has been used in multiple application domains including some scientific workflow systems.

[1] messenger RiboNucleic Acid that mediates the transfer of genetic information from the cell nucleus.

[YB04] proposes a workflow enactment engine (WFEE) with a just in-time scheduling system, which uses a tuple space for exchanging events related to the scheduling of workflow tasks and their execution status on grid resources.

[LP06] presents Rudder, an agent-based system for composition of grid services supporting workflow execution using Comet [LP05]. It provides a decentralized associative shared space, which supports abstractions for workflow composition and execution. Namely it supports a global persistent space for registering and discovering grid services and local spaces as a communication mechanism to support the workflow execution.

[MWL08b] argues that the tuple space model is suitable for executing workflows specified as Petri nets.

DVega [TC09] also uses a specific class of Petri nets, named *Reference nets*, for composing workflows and a tuple space for supporting the interaction between the workflow tasks and the resources for their execution.

[Bal+14] presents a decentralized workflow engine based on the Higher-Order Chemical Language (HOCL) and a shared tuple space for supporting the coordination model inspired by the chemical programming model [CNP08].

✧ **Hierarchies and subworkflows**

Workflows can have dozens or hundreds of activities making it difficult to draw a graphical representation or even the workflow specification using a single level of description.

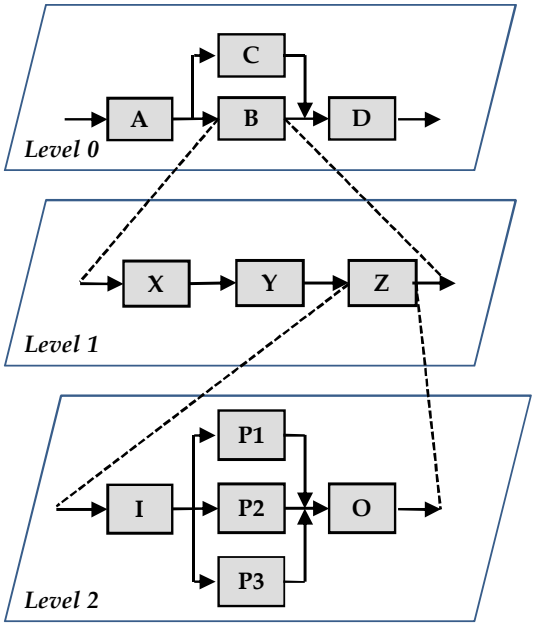


Figure 2.6: Workflow hierarchy

When workflows become more complex, hierarchies become important to keep the

workflow comprehensible [Hoh06]. Thus workflows should be expressed using hierarchies where an activity represents a subworkflow. For instance, Triana (*group units*) and Kepler (*composite actors*) can be used for encapsulating subworkflows as single workflow activities.

In Figure 2.6, the workflow is composed of three levels. At level 0 the *B* activity is a subworkflow represented at level 1, where the *Z* activity is a subworkflow represented at level 2.

2.4.2 Distributed Workflows

The term "distributed workflow" has been ambiguously used in many works and their related publications.

⊸ What is the meaning of distributed workflows?

Sometimes the term "distributed workflow" is used to discuss how much the control of the execution engine is decentralized, and at other times the term is used to indicate that workflow activities can use distributed resources, for instance, the activity tasks can be executed on remote computing nodes although the execution engine is centralized.

This ambiguity already existed in business workflows and naturally has been continued in scientific workflows. In fact, most of the business workflow systems have a centralized execution engine, typically according to the standard architecture defined by WfMC [WfM11]. Also most of the widely used scientific workflow systems have a centralized execution engine implemented to be run as an operating system process. Furthermore, despite the use of a centralized engine, the adoption of service-oriented architectures based on standards, for instance, Web Service Definition Language (WSDL) and Business Process Execution Language (BPEL), has supported the design of workflows as a composition of services executed on distributed resources, which led to consider these workflows as distributed workflows. The Taverna workflow system is a notorious case, where its centralized execution engine orchestrates the invocation of Web Services according to the workflow specification [Wol+13]. Other works rely on particular approaches for using large distributed resources, for instance Circulate [BWH09] is a workflow system for executing data-centric scientific workflows, maintaining the simplicity of a centralized orchestration engine extended with external proxies as gateways, allowing distributed Web Services to exchange data directly, and avoiding data transfers and possible bottlenecks through the centralized engine.

Triana also incorporates modules, such as Grid Application Prototype (GAP) and Grid Application Toolkit (GAT) [Chu+06; Tay+03] for integrating existing grid services and Web Services, as well as peer-to-peer communication allowing remote service invocations. Distributed execution of Triana workflows relies on a specialized *unit* for distributing any task or group of tasks as subworkflows. However, the Triana core processing *units* as well as *units* for data transformation and visualization, are executed by a centralized

enactment engine called Triana Controlling Service.

In Kepler [God+09] distributed workflow execution is achieved by using *composite actors* and by extending *directors* for transporting and executing workflows and subworkflows across a distributed set of computing nodes in order to improve execution performance. In [Plo+13] a variety of distributed execution techniques is presented for Kepler workflows including a distributed data parallel framework using Hadoop [Apa15a] and grid execution by using specific *actors*, such as Serpens, Nimrod/K and Globus.

However, both Triana and Kepler rely on a centralized control for workflow execution. Furthermore the above described Triana and Kepler functionalities for distributed execution of workflows were not yet available as of December 2015 in the downloadable system versions. Therefore end users can not take advantage of these functionalities. This is more noticeable in Triana version 4 [Tri15].

✧ **Decentralizing the control of the workflow execution**

The support for business processes across multiple companies without using a centralized entity is a complex issue and has been an important area of research. This is also true for decentralizing the control of business workflow execution [MWL08a; NCS04].

Therefore, multiple works have investigated and proposed business workflow systems for decentralizing the control of workflow execution. In [HPA05] an autonomic workflow execution engine is discussed and implemented as an extension to the JOpera workflow engine [PA04]. The system is initially deployed as a centralized engine and gradually evolved to a distributed engine using a tuple space used to coordinate request and notification events between Navigators (an extension of the workflow engine) and distributed dispatchers (Task executors). [LMJ10] describes NINOS, a BPEL-based workflow engine for distributing business process execution across several agents where each agent is a single BPEL workflow activity. NINOS utilizes and extends the communication capabilities of the PADRES [Li+06] publishing/subscribing routing infrastructure, allowing the process coordination among the agents. [SS13] presents the OSIRIS-SR system, a distributed workflow engine that relies on a decentralized ring overlay as reliable data storage, allowing a decentralized execution of OSIRIS-SR nodes to autonomously control the execution of workflow instances as a peer-to-peer workflow execution engine.

Also, in the scientific workflows area, some works have proposed models for decentralizing the control of the workflow execution. However, most of these proposals are theoretical models or concrete implementations are not available. As an example, [NPP06] proposes and describes the semantics of a decentralized coordination model based on the chemical metaphor for modeling the execution of workflows. The model envisages that the possible matchings between resources and workflow activities can be modeled by chemical properties defining the affinity of molecules to react. Also inspired in chemistry, [FTP11] presents a decentralized architecture with an external storage (Multiset) as a shared space between workflow activities encapsulating a chemical execution engine where reactions consume data molecules for producing new molecules in an implicitly parallel, autonomous, and decentralized way.

TiDeFlow [Oro+11] is a data-flow execution model for many-core architectures with shared-memory configurations. The model uses *actors* for representing parallel loops expressed as *actor* properties, such as the number of iterations in the loop and a function that contains the code to be executed in each iteration. When an *actor* finishes its execution, it signals other *actors* by sending tokens through a global shared memory queue. The TIDeFlow runtime system is distributed because no one processor, thread, or task is responsible for scheduling, but it is still centralized from the point of view of access to the shared memory. Thus TiDeFlow cannot support parallel computations on distributed infrastructures.

Despite the multiple initiatives for supporting parallel and distributed execution of scientific workflows, there is still a lack of support for an adequate integration with grid and cloud infrastructures. Most of the existing scientific workflow systems have been adapted to support multithreading on multicore architectures. However, few workflow systems are able to incorporate some powerful characteristics of cloud infrastructures like dynamic and autonomous allocation of machines [BL13].

Furthermore the large scale of resources available on grid and cloud infrastructures allows developing scientific workflows executed as a large number of distributed activities, demanding workflow systems with a decentralized control of the workflow execution.

2.4.3 Dynamic Workflow Reconfiguration

The possibility to reconfigure dynamically computational systems during its execution [AC03a] has been studied since the early eighties in multiple contexts, such as distributed systems [KM85], operating systems [Sou+04], software engineering architectures [Bra+04], [KM09].

The goal to support flexibility by adaptation in business workflows has been a concern for several decades [DR09; HSW01; Hei+99]. In fact, in order to keep up the global marketplaces, companies continuously need to introduce rapid changes in business processes and consequently in the workflows used for modeling these processes. Therefore in the last decades a significant research work has been performed in business workflow systems towards supporting workflow adaptation and dynamic changes.

[Hei+99] discusses a classification scheme concerning flexibility of the workflow management and its partial evaluation on a centralized workflow execution engine. The scheme proposed addresses two concepts: flexibility by selection, where modifications of a workflow instance can be done during workflow execution for supporting situations that can not be anticipated during workflow modeling; and flexibility by adaptation, where a workflow must be adapted in order to support new requirements of the business process.

[RD97; RD98] present a framework based on a set of minimal operations for supporting workflow structural dynamic changes, such as insertion and deletion of workflow activities, in the context of a centralized business workflow management system called

ADEPTflex. Issues related to enforcing consistency and correctness are discussed by the same authors in [RB07; RRD04].

[WRR08] describes change patterns and change support features in business workflows, identifying a set of possible changes for adapting the business processes.

However, even considering business workflows many issues still remain open, regarding the support for dynamic changes [APS09; BL10; SJ09].

The scientific workflows also require flexibility for supporting dynamic changes. In fact, scientific experiments modeled as long-running workflows are often designed based on incomplete knowledge on the problem domain. Therefore, by observing the workflow intermediate data or by identifying the need to increase the execution performance during the workflow execution, there is a requirement to support dynamic workflow reconfigurations.

[Liu+07] argues that science experiments require scientists to put efforts on exploring problem-solving methods instead of detailed considerations related to resource management or other implementation details. The work identifies several issues to be addressed in order to achieve an adequate scientific workflow system, such as: i) The granularity of the basic elements and their relationships in a workflow, especially for an unsolved problem; ii) Due to the indeterminism and dynamism of science processes, scientific workflow systems must assist scientists in implementing workflow modifications during execution; iii) What are the soundness properties and the corresponding issues for verifying the consistency of the dynamic modifications.

Some works related to scientific workflows have proposed approaches for improved flexibility in several dimensions of the workflow execution. For example, there is intensive research related to the flexibility of the workflow tasks scheduling.

In the context of Pegasus DAG workflows and the difficulties for mapping the entire workflow before its execution, [Dee+05] proposes a strategy for dynamically mapping the workflow tasks to the available computing resources according to phases of the workflow execution. Each phase is called a future horizon that can be expressed as a set of workflow tasks to be executed within a future predicted time interval. [MFPF12] also discusses mechanisms for scheduling scientific workflows in a commercial multicloud environment using optimization techniques, namely based on monetary costs.

[TCBR11] proposes dynamic runtime adaptive parallelism using Petri nets based patterns for workflow specifications and a case study of a pipeline version of a Montage workflow. The expressiveness of Reference nets (a particular class of Petri nets) allows representing workflow structures with parallelism that can be dynamically modified at runtime.

Based on the chemical computational model [NPP06], in [CNP08] a chemical workflow engine is proposed for supporting dynamic changes according to the change patterns presented in [WRR08].

However, the most widely used scientific workflow systems, such as Pegasus [Dee+05],

Triana [Chu+06], Taverna [Oli+06] and Kepler [Lud+06] have great limitations for supporting dynamic workflow reconfigurations, whether they are for changing the workflow structure or for changing the activities behavior.

Therefore, we identified important open issues related to the support for dynamic changes during workflow execution, mainly if the control of the workflow execution is based on decentralized models. Due to the challenges involved, these issues require new approaches beyond the current state of the art. Thus, we consider that the support for dynamic reconfigurations during the execution of long-running workflows represents a necessary research direction towards improving scientific workflow systems and associated tools.

2.4.4 Failures and Exceptions in the Workflow Execution

Faults and unexpected exceptions originated at workflow, middleware and resource infrastructure levels are usually handled by separate mechanisms in the workflow engine or runtime environment [Gil+07; HA00; HK03; Kam+00]. A taxonomy for faults in scientific workflows is discussed in [Lac+10]. However, the existing approaches for recovering from faults only address very specific concerns for restricted scenarios and well-known and expected situations whose handling is predefined at development time. For example, in some of the more popular scientific workflow systems, such as Taverna [Tav11], Triana [Tri11], or Kepler [Kep13], although there is some support for user handling of faults, the allowed recovery actions follow strategies, such as retrying workflow tasks a certain number of times, or replacing them at runtime, by considering the selection of alternative handling candidates defined at development time, only applying them to well-defined and expected event types.

Other approaches are also restricted to predefined strategies at workflow design time using, for example, workflow patterns [RAH06; TC+10] for dynamic replacement of subworkflows on the occurrence of exceptions.

Therefore, the support for fault and exception recovery remains an open issue, which is still more complex when we consider a decentralized workflow execution engine.

2.4.5 Interactive and Steering Workflows

Dynamic steering of long-running scientific workflows by users is an important requirement. Therefore, workflow systems need to provide facilities for human interactions in order to support workflow monitoring and possible dynamic workflow reconfigurations performed by multiple users.

Computational steering [JPW00; VS99] is the ability to monitor the progress of a long-running application by tracking intermediate data and the possibility to change or calibrate the future behavior of the application. This can be made by the application users by changing the application parameters, by selecting alternative algorithms or even by restarting the application execution with a new configuration. For example,

introducing application decomposition at runtime under user control for load balancing can be considered a form of computational steering to improve performance.

[SKD10] discusses a conceptual architecture and an implementation based on BPEL and the Pegasus workflow system, combining existing scientific and business workflow technologies in order to support human interactions in the management of workflow execution. The work concludes that, unlike business workflows, the integration of human tasks into the execution phase of the scientific workflows was difficult and required a lot of additional effort to develop the appropriate adapters according to the Application Programming Interface (API) provided by the scientific workflow system.

Some large scientific experiments require the processing of distributed data that are stored on multiple repositories [Ben+12], involving multiple users. [Ngu+15] proposes the WorkWays science gateway that supports human-in-the-loop scientific workflows by allowing users to insert or export data in a continuously running workflow.

The vision for supporting user-steered scientific workflows is an important issue towards enabling distributed and collaborative scientific research on many science domains [Gil+07], and is driving significant research efforts [Mat+13]. A survey of the early and current efforts addressing issues in dynamic steering of scientific workflows is presented in [Mat+15].

However, the most widely used workflow systems do not yet provide steering facilities.

2.5 Chapter Summary

Workflows have been used for developing scientific applications in several domains [Chi+11; Dee+05; Tay+07; YB05]. Such efforts have been supported by multiple workflow tools [Laz11], such as Triana [Tri11], Taverna [Tav11] and Kepler [Kep14]. However, there is still a need for improving the support provided by the workflow tools, as discussed for example in [Chi+11; Dee07; Dee+08] and [AGC09]. However, despite the extensive work in the area there are few available scientific workflow systems supported by effective working environments that can be used for developing scientific applications.

The survey presented in this chapter concerning the state of the art of scientific workflows, allows us to reach a paradoxical observation.

On one hand, the increase of e-Science initiatives and the availability of virtualization technologies on cloud infrastructures have created a real possibility for developing scientific applications with high demands of processing throughput. Furthermore, in many scientific domains, modeling these applications by using the workflow paradigm has shown a great potential.

On the other hand, the concept of scientific workflows has shown enormous promises and expectations that have not yet been fully achieved. In fact, existing workflow systems and tools have not yet fulfilled the following issues:

- A decentralized execution control of the workflow activities in order to better take

advantage of the available large distributed computing infrastructures, namely clusters and clouds;

- Decoupling, as much as possible, the workflow execution system from proprietary platforms. In fact, due to the lack of standards in virtualization technologies and cloud infrastructures there is a need for transparent, flexible and portable workflow approaches in order to support their execution on distinct platforms with minimal efforts;

- The support for continuous steering and dynamic reconfiguration in order to make changes to ongoing experiments. This is due to the increased complexity of *in silico* experimentation, which requires long-running workflows that need adaptations during their execution.

This chapter presented an overview based on our systematic study of the intensive research and the large number of publications in scientific workflows. This led us to identify multiple open issues, which encompass the subset of issues addressed by this dissertation.

CHAPTER 3

The AWARD Model

The characteristics of the AWARD model in two perspectives. The programmer's view and the operational view.

This chapter presents the AWARD (*Autonomic Workflow Activities Reconfigurable and Dynamic*) workflow model. Firstly in Section 3.1 the rationale for the model is presented covering a set of issues considered important to support real scientific application scenarios. This is followed by a summary of the main requirements for the design of the AWARD model in Section 3.2. Then the AWARD model is presented according to two complementary points of view.

On one hand, in Section 3.3 we present the workflow developer's view by describing the main characteristics of the AWARD model. In sections 3.3.1 and 3.3.2 we discuss what a workflow developer needs to know in order to define a real workflow specification. In section 3.3.3 we illustrate a complete specification of a simple concrete workflow.

On the other hand, in Section 3.4 we present the AWARD machine as an operational view of the AWARD model by describing the internal operation of an *Autonomic Controller* which supports the model for executing AWARD workflows. The operational view of the AWARD model is based on the Kahn's process networks (PN) model of computation where each workflow activity is supported by an *Autonomic Controller*. The life-cycle of each autonomic workflow activity (AWA) is executed by an *Autonomic Controller* described as a *State Machine* supported by a *Rules Engine*. This allows launching and executing the workflow activities separately on single computers or on distributed infrastructures. The workflow activities exchange information by passing tokens over links connecting activity input and output ports. Links are supported by a communication abstraction named AWARD Space, which allows storing and retrieving tokens during workflow execution.

Section 3.5 presents the chapter conclusions.

3.1 Rationale of the Model

The workflow paradigm was introduced in the second half of the 20th century as a way to organize and automate steps of work within organizations. Initially mainly used in manufacturing companies as a way to increase productivity and ensure quality the workflow concept quickly extended to companies of the most varied businesses. In the last decade of the 20th century, information technology companies gradually aimed at developing software systems based on standards, under the generic name of *Workflow Management System*, enabling organizations to model, run and monitor their business processes as sequences of tasks, arranged as workflows. Also, in the world of scientific computing the developers increasingly adopted the use of the workflow paradigm as a way to model the increasing complexity of scientific experiments that require flexible ways to compose multiple tasks and use large data sets located in different geographical regions.

The similarity between business workflows and scientific workflows [BG07] as a way to run a task graph allowed certain scientific workflows to be modeled and executed using workflow management systems initially developed to support business processes.

However, there are important key issues required to support the development of scientific applications that are not well supported in business workflow management systems and in most of existing scientific workflow systems.

Among the open issues that were surveyed in Chapter 2, this dissertation addresses the key issues described in the following subsections.

3.1.1 Decentralized and Autonomic Workflows

Business workflows systems based on standards established by the Workflow Management Coalition [WfM11] are usually based on centralized execution engines that control the sequence and state of the workflow tasks execution in a monolithic way. In the same way in the world of scientific workflows even the most commonly used systems, such as Triana, Kepler and Taverna (as discussed in Section 2.2) rely on a centralized execution engine responsible for deciding at each time which tasks can be running.

However, centralized approaches pose several difficulties, such as: i) They are not able to model workflow execution involving distributed infrastructures across several geographical regions; ii) They can not support multiple users cooperating on the development and control of workflow activities, where each user can separately execute and control distinct distributed workflow activities; and iii) They preclude the development of large-scale workflows involving an increasing number of activities.

In spite of the above limitations, some centralized systems have been extended to allow a task accessing remote services or launching jobs on remote locations for instance by invoking Web Services or by submitting remote jobs to a cluster scheduler.

Therefore we emphasize the need for a model where the execution of each workflow activity is independent of any form of centralized control. For example, the workflow activities should be able to start/stop separately and run on distributed infrastructures, such as clusters or clouds. This allows running a large number of activities within multiple computing nodes, which could be standalone computers on a local network, multiple cluster nodes, or multiple cloud virtual machines.

Furthermore each activity must have autonomic characteristics according to the autonomic computing vision [KC03] where an autonomic system has self characteristics (self-configuration; self-optimization; self-healing and self-protection), allowing the system to make decisions on its own, based on high-level policies and to dynamically check its status in order to automatically adapt itself to changing conditions.

3.1.2 Transparency and Ease of Programming

While business workflows tend to be developed by specialized software engineers, scientific workflows are often developed by scientists of multiple science domains, without expertise in information technology.

Therefore, a workflow model must provide a sufficient degree of transparency in order to ease the development of scientific applications. The workflow developer should only need to have a minimum knowledge and concern about the low-level operational and implementation details of the underlying workflow execution engine and the execution environment.

In the context of our work an adequate degree of transparency should ideally encompass the following dimensions:

1. At the level of specification of the workflow activities, by decoupling the activity task from the interconnection links between activities allowing to manipulate the workflow structure in an orthogonal way from the task definitions;

2. At the level of the semantics of the interconnection links, by ensuring their independence from the implementation details. For example the tokens that support the flows between workflow activities should be application dependent and not obscure data types compromised with the specific implementation details of the execution engine. If that is not ensured then in some application scenarios it can be difficult to map the application data types to tokens as defined in the execution engine;

3. At the level of the workflow structure, by ensuring that the number of activities and their interdependencies are completely decoupled from the mappings onto the concrete execution environments. That is, the workflow structure should be completely independent from the available computing nodes and the physical data distribution patterns;

4. At the level of the workflow behavior, by separating the functional specification of each activity task from its implementation as an executable unit, such as a thread, an operating system process, a Web Service or a remote job.

3.1.3 Expressiveness of the Workflow Model

The workflow model should provide constructs enabling the developer to express the application requirements related to the following dimensions:

1. Concerning the interactions between workflow activities it is important to support both data-flow and control-flow. In some workflow systems [Shi07b] a distinction is made between a data-flow token when it carries data and a control-flow token when it only carries a synchronization signal, for instance to start or to terminate an activity. Ideally the tokens that support data exchange between activities over the links of the workflow graph should be expressed as generic data types, encompassing both control-flow and data-flow tokens. This gives freedom to the developer for defining the semantics associated to the tokens according to the application requirements;

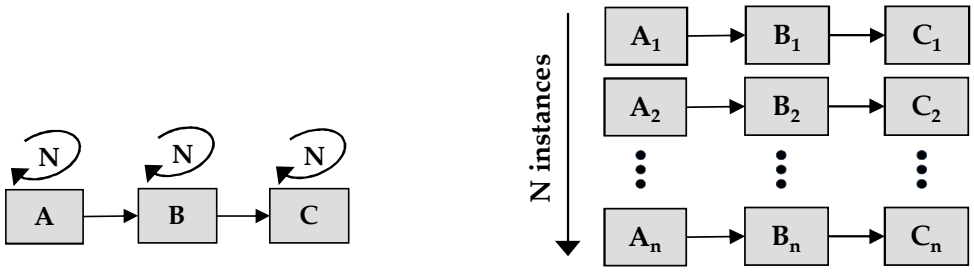
2. Concerning the behavior and the execution control the workflow model should support sequential and concurrent workflow patterns [AHR11; Aal+00b], allowing to express sequences and parallel split and joining of activities with high degrees of concurrency and parallelism. Activities should be able to evolve asynchronously and independently of each other, only requiring synchronization for exchanging tokens;

3. Concerning the type of workflow graphs supported a large number of existing workflow systems only allow direct acyclic graphs (DAG). However, ideally the model should be able to express feedback dependencies or more complex cyclic interactions as loops between the workflow activities;

4. Concerning the workflow specification it is desirable that the workflow model supports the concepts of iterations and instances:

    - **Iteration:** Each workflow activity repeats its execution N times according to the number of iterations as defined by the workflow specification and according to the pace at which input tokens are available;
    - **Instance:** The workflow activities are started N times possibly concurrently according to the number of instances N defined by the workflow specification, each instance possibly having different parameters, and executing independently from the other instances.

The support of iteration and instance concepts introduce distinct behaviors to the execution engine, as discused using the example presented in Figure 3.1.



(a) **Iteration:** each activity processes N tokens, one token per iteration

(b) **Instance:** the workflow is launched N times, spawning N instances of the workflow

Figure 3.1: Workflow iterations and workflow instances

Consider, in Figure 3.1(a), a unique instance of a workflow with N iterations where N is equal to 100. Then the first A activity produces 100 tokens by reading from a data file, then the B activity consumes the 100 tokens as they are produced by the activity A and the C activity produces a result by consuming the 100 tokens as they are produced by the B activity.

One the other hand, in Figure 3.1(b), an example shows the case where a workflow is launched N times generating N independent instances possibly allowing different parameters in some activities in distinct instances, for example in parameter sweep scenarios.

In a workflow with N iterations each individual activity executes its own iterations in a strict sequential order. However, depending on the model the multiple workflow activities can be globally coordinated in different ways across distinct iterations. For example, if the computation model enforces a strict sequential order upon all workflow iterations, then all activities are globally synchronized at the end of each iteration. Therefore all activities must wait for the completion of the $(K-1)^{th}$ iteration before all activities are allowed to start the K^{th} iteration, and then the execution of a workflow with N iterations is equivalent to sequentially launch N instances of the workflow, each with one iteration. As an example, the Synchronous Data Flow (SDF) *director* in Kepler [Kep14] centralizes scheduling decisions in order to enforce a strictly sequential execution order of all iterations.

As an alternative, if the computation model supports independent execution of each activity the iteration concept allows to implement pipelines where multiple items are under concurrent processing, each one in a different activity of the workflow.

Furthermore, the iteration concept has another important characteristic compared with the instance concept that may become important for workflows that must handle a large number of items. For example, if we have 1 million items to be processed, the workflow in Figure 3.1(a) would only need three activities contrasting to the workflow in Figure 3.1(b) that could require 3 million activities.

Nowadays long-term scientific applications can process large data sets, for example using a streaming pattern. This requires a long-running workflow to support the concept

of each activity executing *N*, possibly infinite number of iterations, processing a different data set per each iteration.

Furthermore, a workflow that supports a large number of iterations introduces the possibility to dynamically change the activity parameters facilitating parameter sweep and user steering scenarios.

3.1.4 Dynamic Workflow Reconfigurations

Concerning long-running workflows with multiple finite or even infinite number of iterations the observation of intermediate results of the workflow execution can suggest changes to the workflow structure and/or changes to the behavior of some activities. This motivates an important requirement for a workflow model supporting dynamic reconfigurations by allowing structural and behavioral changes during the workflow execution. It should be possible to change the workflow structure by adding or removing activities and by creating or deleting links. It should also be possible to change the workflow behavior by replacing the algorithm of any activity task, as well as by changing other characteristics affecting activities and links.

3.1.5 Effective Support for Experimentation

In order to enable the practical development and evaluation of the workflow applications it is important to provide operational prototype support to the basic mechanisms required to perform monitoring of the workflow execution by logging all activities information states, including about the tokens exchanged between activities. This also enables the development of debugging and data provenance functionalities allowing future workflow reuse and improvements.

3.2 The AWARD Model Requirements

To address the above key issues we identified the following requirements for the AWARD model (acronym for *Autonomic Workflow Activities Reconfigurable and Dynamic*):

1. **Decentralized and autonomic computational model:** Each workflow activity of the AWARD model should have autonomic behavior for running independently of any centralized control. The workflow activities should be able to start/stop separately for running in parallel according to the Process Networks (PN) model of computation [Kah74]. The *Task* executed by an activity should be driven by tokens received on the activity input ports that are connected to other activities;

2. **Transparency and ease of programming:** The AWARD model should support workflow specifications focused on the application requirements so that the workflow developer only needs a minimum knowledge on the low-level operational and implementation details of the workflow execution engine and the execution computing

environment. Therefore the AWARD model should provide transparency and ease of programming concerning the following characteristics:

(a) The workflow structure specification should be orthogonal, that is independent, from the *Task* definitions;

(b) The activity *Task* should be decoupled from the interconnection links between activities;

(c) The activity *Task* specification should be decoupled from its implementation. A *Task* should encapsulate any kind of executable unit, such as a thread, an operating system process, an invocation of a Web Service, or a remote job;

(d) A clear separation should be promoted between the workflow specification and its mapping to the execution environment. For instance the same workflow specification with minimal modifications to its mappings should be executable on a single standalone computer, on multiple computers on a local computer network, or on distributed infrastructures, such as clusters and clouds;

3. **Expressiveness of the workflow model:** The AWARD model should provide constructs allowing the workflow developer to express the application requirements using the following characteristics:

   (a) The control or data tokens flowing between the workflow activities should have application-dependent semantics and be expressed as generic data types and not obscure data types compromised with the execution engine details;

   (b) The workflow model should support sequential and concurrent workflow patterns for expressing sequences and parallel splitting and joining of activities with high degrees of concurrency and parallelism. Activities should be able to evolve asynchronously and independently of each other, only requiring synchronization for exchanging tokens;

   (c) The workflow specification should support expressing multiple structural layouts, including feedback loops between activities;

   (d) The workflow specification and execution should support multiple iterations or even infinite number of iterations as long-running workflows, where different activities can run at their own pace, each in a different iteration number;

4. **Dynamic workflow reconfigurations:** During the execution of long-running workflows it should be possible to change the workflow structure and behavior by applying dynamic reconfiguration plans affecting only one activity or multiple activities without the need to restart the entire workflow execution. Each activity must be autonomic for applying its part of the reconfiguration plan;

5. **Effective support for experimentation:** The AWARD model must be implementable by an operational prototype enabling the practical development and evaluation of

the workflow applications, including mechanisms supporting monitoring of the workflow execution and tools for launching the workflow execution and submitting dynamic reconfiguration plans.

3.3 The AWARD Model: The Programmer's View

The objective of this section is to present the essential concepts and definitions of the AWARD model allowing programmers to specify their applications as workflows.

The AWARD model is based on Kahn Process Networks (PN) model [Kah74] where processes communicate through unbounded FIFO channels by reading and writing atomic data elements called tokens.

The PN model of computation assumes that the write operation is non-blocking and always succeeds, while the read operation is blocking, that is reading tokens from an empty channel blocks the process until at least a token is available. The PN model does not allow processes to check input channels to test for the existence of tokens without consuming them.

✧ **Autonomic workflow activity (AWA)**

In the AWARD model, a PN process is an *Autonomic Workflow Activity* (AWA) executed inside an independent operating system process so it is possible that workflow activities or groups of activities are launched and executed on distributed computing nodes.

Each AWA activity has a possibly empty set of input ports where the activity receives tokens from other activities, and a possibly empty set of output ports to which the activity sends tokens that are passed to other activities via links established between activities.

The model guiding the composition and execution of activities relies on a decentralized control approach where each AWA activity is autonomic. The coordination of AWA activities is token driven. When all input ports of an activity have tokens to be processed the activity is enabled and immediately starts executing its $Task$, which encapsulates the corresponding application algorithm assigned to the workflow activity. When its $Task$ terminates, the activity produces tokens on its output ports for enabling other activities to start their own tasks. Then according to the PN model, the AWARD model supports a parallel execution of activities where activities proceed asynchronously with a coordination only based on the tokens production pace.

✧ **Links between input/output ports**

In the AWARD model there are not explicit communication channels between input and output ports of AWA activities, that is, the interconnection between activities is only implicitly specified through the relationship defined between output and input ports. The connection or link between an output port and an input port is based on a communication abstraction with the following properties:

1. **Direct communication:** Each output port explicitly specifies a list of input ports as communication destinations;

2. **Anonymous communication:** In order to get an input token, the receiver input port does not have to know the identification of the emitters (source) of the data received;

3. **Unbounded links:** Each link connecting an output port to an input port supports an unbounded number of tokens allowing different paces of producing/consuming tokens.

✧ **Tokens**

A token is composed of information related to exchange data or control between activities and, when it is emitted by an output port is marked with the name of the destination input port.

✧ **Activity *Tasks***

Each AWA activity has an associated *Task*, which is a software component developed without the need to be aware of details on any execution engine. A *Task* is developed as a Java class implementing a generic interface specifying an entry point function where the *Task* receives as input an unbounded number of objects and returns as output an unbounded number of objects. The *Task* internal definitions are only dependent on the application algorithms. Then the *Task* is able to encapsulate calls to externally defined services, such as Web Services, or run local or remote programs, supporting binary legacy applications in other languages, for example in C or Fortran.

The objects that implement the activity *Tasks* are bound dynamically. This occurs when an activity needs to execute its *Task*. Then the activity instantiates an object that implements the generic interface and calls the entry point function. In such a way the activity specification only needs to know the absolute name of the class that implements the activity *Task*.

✧ **Iterations and instances**

The AWARD model supports the concept of workflow iterations. Therefore the specification of an AWARD workflow defines a global maximum number of iterations. By default each AWA activity executes a number of times according to this maximum number of iterations. However, each AWA activity can override that maximum number of iterations by specifying a different number of iterations in the individual AWA specification. The AWARD model is flexible to support different types of iteration models: i) A simple Direct Acyclic Graph (DAG) workflow where the number of iterations is one; ii) An infinite number of iterations, where each activity runs continuously by consuming tokens on its inputs and producing tokens on its outputs; and iii) An activity can specify a specific number of iterations. For example in a scenario where an activity consumes, in each iteration, any token produced from multiple activities, the consumer activity needs to run as much iterations as the sum of the iterations of the producer activities.

The AWARD model supports the concept of instances as distinct and possible concurrent workflow executions. However, the AWARD model does not support any interdependency between instances.

✧ **Flexibility and expressiveness**

The properties of the links, namely the anonymous communication, introduce flexibility in the workflow specification by decoupling an activity from all activities that are upstream in the workflow graph.

Since each token is marked with the unique name of the destination port, the communication abstraction for links can be mapped to any data store paradigm supporting data storing and data retrieving operations using primary keys or associative memory.

According to each application scenario and the corresponding interdependencies between activities, the workflow developer can decide on the semantics of the tokens production and consumption, repectively at output and input ports.

✧ **Basic workflow patterns**

The AWARD model supports the expressiveness of sequential and concurrent workflow patterns [AHR11; Aal+00b], such as sequence, parallel split and parallel merge, as illustrated in Chapter 6 of this dissertation. The model also supports dynamic reconfiguration scenarios, as presented in Chapter 4.

The data parallelism pattern with independent activities, as depicted in Figure 3.2, can be expressed as an AWA activity whose name is A with multiple output ports connected to input ports of distinct $(P_1,...,P_n)$ AWA activities executed in parallel. When the A AWA activity puts tokens in its outputs ports, the $(P_1,...,P_n)$ AWA activities are enabled to start their execution independently.

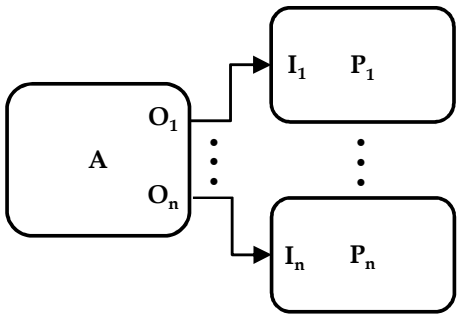


Figure 3.2: Data parallelism pattern

The merge pattern can be expressed as depicted in Figure 3.3, where multiple independent parallel AWA activities $(P_1,...,P_n)$ are connected to a merge AWA activity (M) that performs some data merge.

The M AWA activity also performs a synchronization action in each iteration, waiting for a token produced by each of the $(P_1,...,P_n)$ AWA activities that are executed independently with possible different paces of tokens production.

The pipeline pattern can be expressed as depicted in Figure 3.4, where three stages are developed as the A, B and C AWA activities. Data into the pipeline are sequentially processed at each stage. The tokens produced by the A activity are received by the B activity and tokens produced by the B activity are received by the C activity. For a pipeline

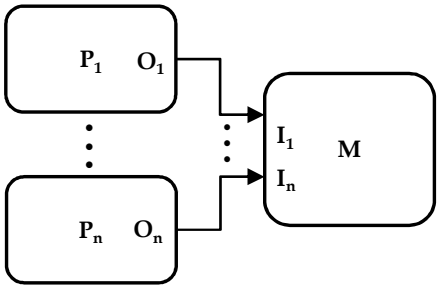


Figure 3.3: Merge/Synchronization pattern

working on data streaming, activities repeat their functionalities. This is supported by defining a workflow with iterations where the pace of tokens production by the *A*, *B* and *C* activities can be distinct, for instance the *A* activity may already have produced *N* tokens, meaning that has completed the N^{th} iteration and the *B* activity may still only have produced *K* tokens with *K* much smaller than *N*.

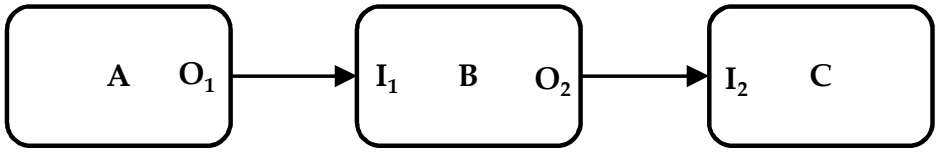


Figure 3.4: Pipeline/Streaming pattern

In the above examples tokens can carry some application-dependent data, for instance, filenames or can only carry control for synchronizing the start of activities.

The programmer's view of the workflow specification is based on the assumption that in the AWARD model the development of application algorithms only requires user's knowledge on their logical decomposition into *Tasks* and the specification of activity dependencies through their input and output ports.

In the following sections we present: i) The main concepts of the AWARD model (Section 3.3.1); ii) A set of definitions that a workflow developer should know to specify AWARD workflows (Section 3.3.2); and iii) The example specification of a concrete workflow (Section 3.3.3).

3.3.1 Main Concepts

In order to explain the main concepts of the AWARD model we use the example depicted in Figure 3.5. This AWARD workflow is a graph with activities named as A, B, C, D and E connected by unidirectional links with the $L_{AC}, L_{BC}, L_{CD}, L_{CE}$ names to exchange the T_1, T_2, T_3, T_4 control or data-flow tokens.

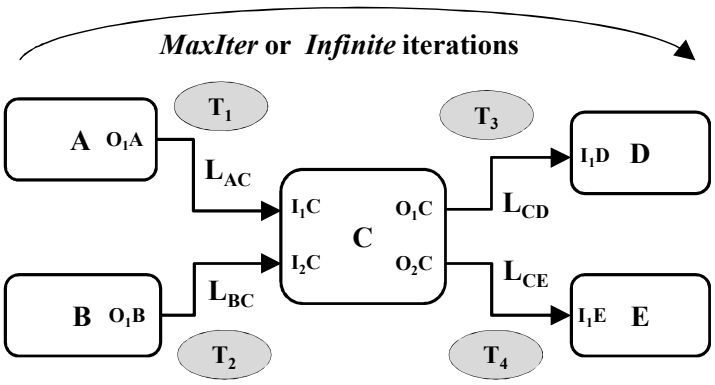


Figure 3.5: Workflow with the A, B, C, D and E activities, connected by $L_{AC}, L_{BC}, L_{CD}, L_{CE}$ links and the T_1, T_2, T_3, T_4 flow tokens

✧ **Links**

Links correspond to associations between activity ports. For example, the C activity in the workflow represented in Figure 3.5, has input ports named I_1C and I_2C and output ports named O_1C and O_2C . The origin of a link is an *Output* port and the link destination, shown as an arrow in Figure 3.5, is an *Input* port.

The AWARD model does not require any explicit definition of the links as separate entities connecting ports. *Links* are implicitly defined by associations between names of output and input ports. For instance, the L_{AC} link in workflow represented in Figure 3.5 is implicitly defined by the pair $(O_1A,\ I_1C)$ where, O_1A is the name of the output port of the A activity and I_1C is the name of an input port of the C activity. An activity can have multiple input and output ports, including the possibility to have only outputs (A and B activities) or only inputs (D and E activities).

✧ **Iterations**

An AWARD workflow has always an associated global iterations number which defines a maximum number for the workflow iterations, which must be greater than zero. By default this iteration number is inherited by all workflow activities. However, each workflow activity can specify an individual maximum number of iterations specific to the activity. The maximum number of workflow iterations is denoted by $MaxIter$, if it is finite, or $Infinite$ for long-running workflows performing iterations continuously.

An activity terminates its execution after it performs the $MaxIter^{th}$ iteration. If a workflow has an $Infinite$ number of iterations, the activities can only terminate upon receiving a command for explicit forced termination.

✧ ***Task***

A workflow activity has an associated functionality defined by its *Task*, which receives as input an array of *Arguments* containing data resulting from the mapping of the input ports and an array of constant activity *Parameters* defined in the workflow specification. After executing the *Task* algorithm with the input *Arguments* and *Parameters*, the data returned by *Task* are mapped to the activity output ports.

✧ **Tokens**

Each AWA activity can produce tokens on its output ports at its own pace. The execution of each AWA is only driven by the availability of tokens in all input ports of the AWA activity, independently of other activities.

As an example, depending on the global number of iterations of the workflow of Figure 3.5 on page 56, the *A* and *B* activities produce a certain number of tokens, respectively, T_1 and T_2. As the *A* and *B* activities proceed asynchronously, it may happen that the *A* activity can produce all of its tokens even before the *B* activity produces any of its tokens. In any case, the *C* activity will only be enabled to proceed each time whenever both tokens from *A* and *B* are available.

The AWARD model ensures that a *Link* supports an unbounded set of tokens. This allows the workflow activities to produce and consume tokens at different paces according to their execution speed per iteration.

The AWARD model also ensures that there is no loss of tokens, that is, *Links* are reliable in the sense that each produced token will be consumed providing that the destination activities are not failed or terminated before the maximum number of iterations, ($MaxIter$), is reached.

3.3.2 Workflow Specification

From a point of view of a workflow application developer, the specification of AWA activities has two views:

1. An external view of the entire set of AWA activities and their interactions is related to the topological structure of the workflow and the corresponding implicit links between activities;

2. An internal view of each single AWA activity is related to the implementation of the activity $Task$, the state and operation of the input and output ports as well as how activity *Parameters* and tokens from input ports are mapped to the $Task\ Arguments$ and how $Task\ Results$ are mapped to tokens on output ports of the activity.

Together both views are required to complete the workflows specification. Then the specification of AWARD workflows requires knowledge about the following concepts:

✧ **Naming**

A workflow, each workflow activity, each input port and each output port have global names, assumed to be unique.

Definition 3.1: ***Naming***

Unique names are defined by the elements in a set $Names = \{name_1, ..., name_n\}$ where each $name_i$, $1 \leqslant i \leqslant n$, is a sequence of characters and each character belongs to the set $Characters = \{A...Z, a...z, 0...9, -\}$.

✧ **AWARD workflow**

An AWARD workflow (W) is defined as a tuple with a workflow name ($Wname$), a global number of maximum iterations ($gIterations$) and a graph (G) of activities.

Definition 3.2: ***AWARD workflow***

An AWARD workflow is defined by a tuple $W = (Wname, gIterations, G)$, where: $Wname$ is the workflow name; $gIterations$ is a number greater than zero denoted by $MaxIter$ or is denoted by the $Infinite$ keyword meaning that the workflow has an infinite number of iterations; and G is the workflow graph (as in Definition 3.3).

Definition 3.3: ***AWARD workflow graph***

The graph G of the workflow is a tuple $G = (Activities, Links)$ with a finite non-empty set of activities ($Activities$) and a set of unidirectional links ($Links$) defining the connections between output and input ports.

✧ **Tokens**

The information flow between workflow activities is based on token passing over the links that connect the activities. A token is composed of two types of information concerning exchange data or control flow between activities: i) Data information to be mapped to the $Arguments$ of the activity $Task$; ii) Control information to be used for controlling the execution of activities.

Definition 3.4: ***Token***

A token T transmitted through a link is defined by a tuple $T = (vT, I, Seq, destInName)$. The field vT is data value information according to an application-defined data type. The control information placed by the emitter activity consists of two fields, I and Seq. By default the iteration number (I) is defined as the current iteration number of the emitter activity. However, if the activity has an input port in the $Sequence$ input mode (Definition 3.7) then the iteration number (I) in the produced token is equal to the iteration number contained in the token received in the input port. The Seq field is a sequence number defining the total ordering of tokens produced in each link and it can be used to control the reception ordering of the tokens on the downstream destination port. The field $destInName \in Names$ is the name of the destination input port.

In the above Definition 3.4, the data type of the token value vT is a primitive or an application-dependent data type, denoted by $Ttype$. For instance, for a $Ttype = Integer$ we can have $vT = 5$, or for an object value of the type $Ttype = SomeClass$ we can have $vT = new\ SomeClass()$ where vT is assigned as an object instance of the $SomeClass$ class.

✧ **Activity**

Each activity (A) is defined as a tuple with an activity name ($Aname$), a possibly empty array of constant $Parameters$ ($APars$), a possibly empty set of input ports ($AInputs$), a possibly empty set of output ports ($AOutputs$), the name of the activity $Task$ ($ATask$), a mapping from input port names to $Task\ Arguments$ ($AMapins$), a mapping from $Task\ Results$ to output port names ($AMapouts$) and an optional number ($AMaxIter$) specifying an individual maximum number of iterations specific to this activity, which overrides the global workflow maximum iterations number ($gIterations$).

Definition 3.5: ***Activity***

An activity (A) is defined as a tuple
$A = (AName, APars, AInputs, AOutputs, AMapins, ATask, AMapouts, AMaxIter)$.

✧ **Inputs and outputs**

Each workflow activity can have zero or more input and output ports.

In each iteration, each workflow activity consumes one token from each of its input ports and produces one token for each of its output ports. According to Definition 3.4 each token carries an iteration and a sequence number, which are used to determine the order of token consumption at each input port and that is equivalent to a FIFO discipline in the links as in the PN netorks model.

Nevertheless, we have decided to also allow the possibility of handling non-deterministic behavior in a merge workflow activity by defining an input port mode with a non-deterministic semantics (Any mode as in Definiton 3.7). This is found useful in practical situations where unpredictable sequences of events must be handled. Other authors [AB84; BH01; LP95] also argue in favor of a non-deterministic merge.

Definition 3.6: ***Inputs and outputs***

An activity (A) has a set of input ports $A_{Inputs} = \{I_1A, ..., I_nA\}$ and a set of output ports $A_{Outputs} = \{O_1A, ..., O_nA\}$ where I_iA represents the name of the i^{th} input port of the A activity and O_iA represents the name of the i^{th} output port of the A activity.

✧ **Input port**

An input port of an activity (A) is defined by a tuple, named the *input port configuration context* ($CfCtx_{in}$) as in Definition 3.7.

✧ **Output port**

An output port of an activity (A) is defined by a tuple named *output port configuration context* ($CfCtx_{out}$) as in Definition 3.8.

Definition 3.7: *Input port*

The input port configuration context is defined by a tuple
$CfCtx_{in} = (InameA, Ttype, IMode, State)$
with the following fields:

- A unique global name ($InameA \in Names$) (as in Definition 3.1);
- A type $Ttype$, which is the data type of the token value (as in Definition 3.4);
- The input port mode, which specifies how the input port receives tokens ($IMode \in \{Iteration, Sequence, Any\}$). For the $Iteration$ and $Sequence$ modes, the input port consumes tokens respectively according to the total ordering of the iteration or sequence numbers (as in Definition 3.4). For the Any mode, tokens are consumed non-deterministically, in any order irrespective of the iteration and sequence numbers in each token
- In the case of the $Sequence$ input mode, there is a restriction imposing that the activity can only have a single input port, which can only be connected to a unique simple link (as in Definition 3.9).
- The field ($State$), which specifies the initial state of the input port (as in Definition 3.15).

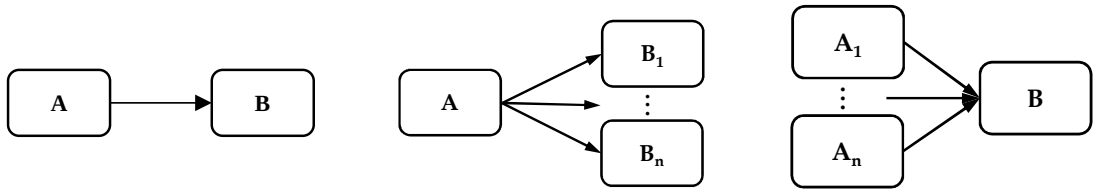
Definition 3.8: *Output port*

The output port configuration context is defined by a tuple
$CfCtx_{out} = (OnameA, Ttype, OMode, SendTo, State)$
with the following fields:

- A unique global name ($OnameA \in Names$) (as in Definition 3.1);
- A type ($Ttype$) that is the data type of the token value (as in Definition 3.4);
- A non-empty set ($SendTo$) of simple links SL_{ink} (as in Definition 3.9) originated in that output port;
- The output port mode ($OMode \in \{Single, Replicate, RoundRobin\}$) which specifies how the output port sends tokens: The $Single$ mode applies when $SendTo$ has a single element and a token is sent to the single destination input port. The $Replicate$ and $RoundRobin$ modes are applied when $SendTo$ has multiple elements (as in Definition 3.10);
- The value ($State$), which specifies the initial state of the output port (as in Definition 3.15).

✧ ***Links***

The workflow specification does not require an explicit definition of the *Links* as separate entities connecting output and input ports. *Links* are implicitly defined by associations between the unique names of the ports, which are specified in the output port definitions.



(a) A simple link connects one output to one input (sequence pattern)

(b) A multi-link connects one output to multiple inputs (split pattern)

(c) A multi-link connects many outputs to one input (join pattern)

Figure 3.6: Links *versus* input and output ports of activities

Definition 3.9: *Links*

A simple link SL_{ink} is an ordered pair $(O_1A,\ I_1B)$ which represents a connection between the O_1A activity output port (the origin of the link) and the I_1B activity input port (the destination of the link) (see Figure 3.6(a)). A multi-link ML_{ink} is a set of simple links SL_{ink} representing an output port connected to multiple input ports (see Figure 3.6(b)) $ML_{ink} = \{(O_1A,\ I_1B1),...,(O_1A,\ I_1Bn)\}$ or a set of simple links SL_{ink} connecting multiple output ports to an input port (see Figure 3.6(c)) $ML_{ink} = \{(O_1A1,\ I_1B),...,(O_1An,\ I_1B)\}$.

Definition 3.10: *Multi-link split pattern*

For a ML_{ink} multi-link, such as in Figure 3.6(b) the origin output port emits tokens in one of two distinct modes ($OMode$ as in Definition 3.8): i) **Replicate**, such as each produced output token is replicated and sent to every connected destination port included in the multi-link; and ii) **RoundRobin** where each produced output token is marked with a sequential number (Seq) for each destination link and is alternately sent in a round-robin fashion to each one of the connected destination input ports included in the ML_{ink} multi-link. In this case each of the destination input ports must be in the $Sequence$ input mode because the iteration numbers marked in each token produced for each link are not consecutive.

Definition 3.11: ***Multi-link join pattern***

For a ML_{ink} multi-link, such as in Figure 3.6(c)) the destination input port of the B activity consumes the tokens received from the connected output ports, that is, the multiple sources of the multi-link.
In this case the input port mode ($IMode$) (Definition 3.7 on page 60) is restricted to the $Iteration$ or Any cases. Due to the B activity consumes a token on its input port for each iteration, in the $Iteration$ mode the B activity consumes tokens sequentially according to the activity current iteration number independently of the source link. In the Any mode the B activity consumes any token without any order. In this case the number of iterations of the B activity must be equal to the sum of the number of iterations of all $A_1 ... A_n$ source activities.

✧ ***Task***

Each activity (A) is responsible for implicitly managing the following sequence of steps, sketched in Figure 3.7:

1. $AGetTokens$ get one token for each of its enabled input ports;
2. $AMapins$ maps these tokens to the $Task\ Arguments$;
3. $ATask$ invokes the AWA $Task$ using the $Arguments$ and the AWA $Parameters$;
4. $AMapouts$ maps the $Task\ Results$ to all its output ports;
5. $APutTokens$ produces one token for each of the mapped output ports.

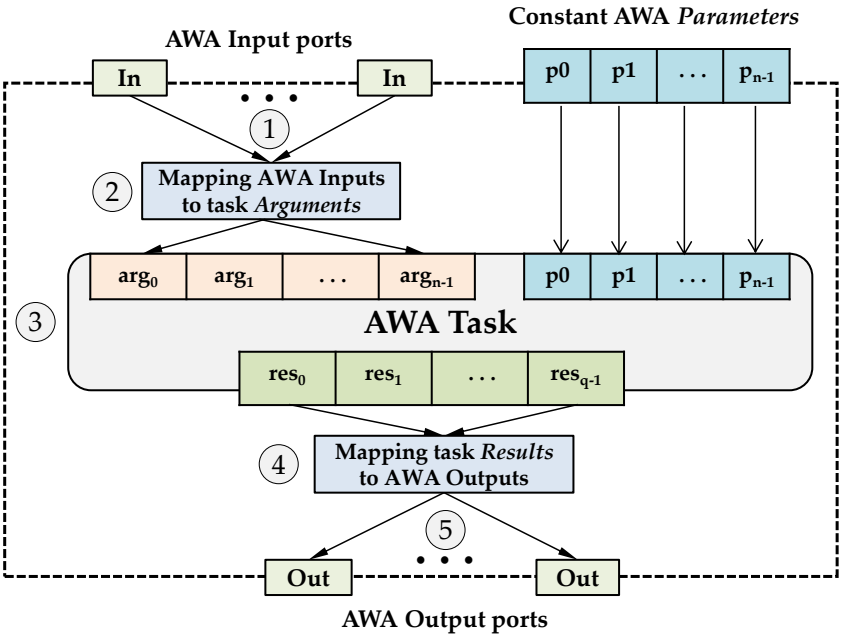


Figure 3.7: The internal view of an AWA activity

Definition 3.12: ***Activity Task***

An AWA *Task* (*ATask*) is an implementation of a generic interface *IGenericInterface* that specifies the *Task* execution entry point, $EntryPoint(Arguments, Parameters) \rightarrow Results$, where *Arguments* is an array of items mapped from the activity input ports $[AT_{\arg s}0, ..., AT_{\arg s}(nargs-1)] = AMapins(A_{Inputs})$, *Parameters* is an array of constant parameters directly obtained from the activity specified field (*APars*) and *Results* is an array of items $[AT_{res}0, ..., AT_{res}(q-1)]$ to be mapped as tokens to the activity output ports $A_{Outputs} = AMapouts(Results)$.

✧ **Mappings**

Definition 3.13: ***Mapping Inputs***

The mapping of input ports to *Task Arguments*, *AMapins*, is defined by a function that maps the activity input port names to the array of the *Task Arguments* and returns a set of pairs where each pair associates at most one position in the *Arguments* array with an input port name.
$AMapins = \{(0, Iname_0), ..., (k, Iname_k), ..., (nargs-1, Iname_{nargs-1})\}$ where
$0 \leqslant k \leqslant n\arg s - 1$ and $Iname_k \in A_{Inputs}$.

Definition 3.14: ***Mapping Outputs***

The mapping of *Task Results* to the activity output ports, *AMapouts*, is defined by a function that maps the array of *Task Results* to all output port names and returns a set of pairs where each pair associates each output port name $Oname_k$, $0 \leqslant k \leqslant N-1$ (where N is the number of output ports) with at most one position in the *Task* results array,
$AMapouts = \{(Oname_0, res_0), ..., (Oname_k, res_k), ..., (Oname_{n-1}, res_{n-1})\}$ where
$0 \leqslant res_k \leqslant nres - 1$ and $Oname_k \in A_{Outputs}$.

✧ **Specification of an AWARD activity**

From the above definitions the programmer's view of the specification of the AWARD activity, named A, with m input ports and n output ports, is described in Equation 3.1.

$$\{O_1A, ..., O_nA\} = AMapouts(ATask(APars, AMapins(\{I_1A, ..., I_mA\}))) \tag{3.1}$$

✧ **Setting up the state of ports and activities**

Definition 3.15: ***The state of input and output ports***

The state of an input or an output port is one of the set $\{Enable, Disable, EnableFeedback\}$ with the following semantics:

1. In the *Enable* state, the port is ready to process the associated tokens (the received tokens for an input port or the produced tokens for an output port);
2. In the *Disable* state, there is a null action on ports: i) For an input port, the tokens associated with this port are not processed by the *AGetTokens* control step of the activity; and ii) For an output port, the tokens are not produced.
3. The *EnableFeedback* state only applies to cyclic workflows with multiple iterations as follows:
   - Given an activity A with current iteration K, all input ports that have the *EnableFeedback* state set are not processed by the *AGetTokens* step at this iteration K. Instead, those input ports are automatically changed to the *Enable* state and will be considered only when the A activity execution reaches the next iteration $(K+1)$;
   - Given an activity A with current iteration K, for all output ports that have the *EnableFeedback* state set, the *APutTokens* control step marks the iteration field I in the produced tokens (as in Definition 3.4) with an iteration number equal to $(K+1)$. In order to avoid the production of tokens that would never be consumed, an output port that has the *EnableFeedback* state set, is automatically set to *Disable* when the activity reaches the iteration $(MaxIter-1)$.

The workflow depicted in Figure 3.8, is an example of a cyclic workflow using the state *EnableFeedback* to manage the ports (O_1F) and (I_1A) involved in a feedback loop.

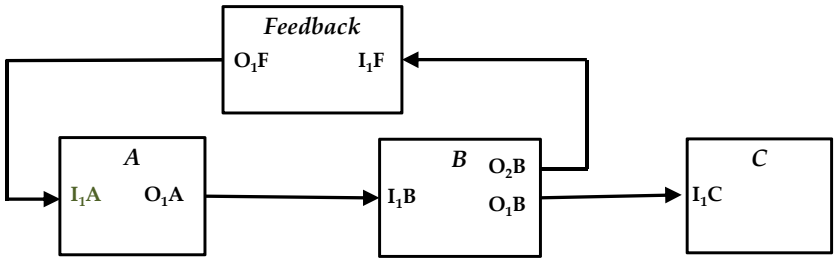


Figure 3.8: Workflow with a feedback loop

This requires that as a result of executing the K^{th} iteration in the *Feedback* activity its output port (O_1F) in the *EnableFeedback* state must produce tokens to be consumed only at the $(K+1)^{th}$ iteration by the connected input port (I_1A) in the A activity. The (I_1A) port

is set to be in the *EnableFeedback* state at the K^{th} iteration, passing automatically to the *Enable* state at the $(K+1)^{th}$ iteration in the A activity, in order to consume the token sent from the (O_1F) output port.

According to the specification of the activity input ports state, the condition for an AWA activity execute its *Task* is presented in Definition 3.16.

Definition 3.16: ***Condition for an activity to execute its Task***

An activity is enabled for executing its *Task* when all its enabled input ports have received tokens to be processed. If an activity has no input ports, the execution of its *Task* is always enabled.

3.3.3 Specifying a Concrete Workflow

To illustrate the specification details let us consider the workflow of Figure 3.8 on page 64 as an example of a data filtering scenario with *Infinite* number of iterations and involving a feedback loop where we assume that the A, B, C and *Feedback* activities have the following characteristics:

- The A activity processes files stored in a directory whose name is passed as a *Parameter* and prepares data sets, whose data type has the *wkf.AppRecord* absolute name, that are then sent via the O_1A output port to the I_1B input port of the B activity;
- The B activity processes the data sets received on the I_1B input port, and produces, on the O_2B output port, an array of strings with features extracted from the data set to be sent to the I_1F input port of the *Feedback* activity. Furthermore the unchanged data set is sent via the O_1B output port to the I_1C input port of the C activity;
- Based on a database with knowledge about historical features, the *Feedback* activity analyzes the array of features received on its I_1F input port and produces, on the O_1F output port, a data filter criterion stored as a data type with the absolute name *wkf.FilterCriteria* which is sent (closing the loop) to the I_1A input port. For each iteration this loop allows the A activity to receive feedback from the previous iteration for improving the data sets produced. The connection details to access the database are defined through an activity parameter;
- The data sets received on the I_1C input port of the C activity are stored in a database with connection details passed as a parameter of the C activity.

The complete workflow specification is presented in Table 3.1 on page 67 where the italic fields are the names of the specification attributes and the bold strings are the specific specification values of the workflow presented in Figure 3.8 on page 64.

The workflow specification has an header with the workflow name and a maximum iterations number (*Infinite*) indicating that each activity performs an infinite number of iterations.

The workflow activities (AWA) are specified as independent blocks where each activity specifies its name, its input and output ports, and its *Task*.

It is important to note the almost complete independence between all AWA specifications. In fact, only the *SendTo* field of the output ports specification carries dependencies between the AWA specifications by indicating the names of the destination input ports of the other downstream AWA activities.

In order to illustrate an example of how to develop an AWA *Task* we present, in Listing 3.1, the skeleton of the *Task* of the *Feedback* activity.

According to the workflow specification (Table 3.1) the programmer responsible for developing this *Task* only needs to know the order and the data types of the *Arguments*, the *Parameters* and the *Results* to be returned from the *Task*.

Listing 3.1: An example of an AWA *Task*

```
1  // Class which implements the Task of the Feedback activity
2  public class TaskFeedback implements IGenericTask {
3    public Object[] EntryPoint(Object[] Arguments, Object[] Parameters) {
4       // From mapping of the I1F input
5       String[] features = (String[]) args[0];
6       // Gets the activity Parameter
7       String DBconnect = (String) Parameters[0];
8       // Analyze the features and produce the filter criteria object
9       wkf.FilterCriteria fcrit= new wkf.FilterCriteria(. . .);
10      // Return only one result as an object of the wkf.FilterCriteria class
11      Object[] Results = new Object[]{ fcrit };
12      return Results;
13   }
14 }
```

This simple example highlights some important characteristics of the AWARD model:

- The workflow specification consists of the specification of a list of AWA activities;
- The interactions between AWA activities (*Links*) are specified as associations between input and output ports;
- *Task* developers do not need to know low-level details about any execution engine focusing solely on the application algorithms.

Table 3.1: Specification of the workflow depicted in Figure 3.8 on page 64

Workflow Name	**Data Filtering Workflow with a Feedback Loop**
Maximum Iterations	**Infinite**

AWA	*Name*	**A**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1A	**EnableFeedback**	**wkf.FilterCriteria**	**Iteration**	
	Outputs	*Name*	*State*	*TokenType*	*OMode*	*SendTo*
		O1A	**Enable**	**wkf.Record**	**Single**	**I1B**
	Task	*Parameter*		**/usr/filteringFiles/**		
		SoftwareComponent		**wkf.TaskProduceRecords**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1A	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				0	**O1A**	

AWA	*Name*	**B**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1B	**Enable**	**wkf.Record**	**Iteration**	
	Outputs	*name*	*State*	*TokenType*	*OMode*	*SendTo*
		O1B	**Enable**	**wkf.Record**	**Single**	**I1C**
		O2B	**Enable**	**java.lang.String[]**	**Single**	**I1F**
	Task	*Parameter*		**Null**		
		SoftwareComponent		**wkf.TaskExtractFeatures**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1B	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				0	**O1B**	
				1	**O2B**	

AWA	*Name*	**Feedback**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1F	**Enable**	**java.lang.String[]**	**Iteration**	
	Outputs	*name*	*State*	*TokenType*	*OMode*	*SendTo*
		O1F	**EnableFeedback**	**wkf.FilterCriteria**	**Single**	**I1A**
	Task	*Parameter*		**"server=S;database=DBHist;"**		
		SoftwareComponent		**wkf.TaskDataFiltering**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1F	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				0	**O1F**	

AWA	*Name*	**C**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1C	**Enable**	**wkf.Record**	**Iteration**	
	Task	*Parameter*		**"server=S;database=DBREC;"**		
		SoftwareComponent		**wkf.TaskStoreRecords**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1C	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				Null	**Null**	

3.4 The AWARD Machine: The Operational View

The AWARD machine follows a distributed model of computation based on the Kahn process networks (PN) [Kah74]. The workflow activities (PN processes) communicate through *Links* as abstractions implicitly defined by associations between the unique names of the activity ports (as in Definition 3.9). *Links* connect output ports to input ports for passing *tokens* (as in Definition 3.4).

✧ **The computation model of the AWARD machine**

The AWARD model is based on the Process Networks (PN) model, where processes are named *Autonomic Workflow Activities* (AWA). Each AWA activity is executed within an independent operating system process running an *Autonomic Controller* (AC) which controls the life-cycle of the AWA activity. This model of computation allows the workflow activities (AWA) to be launched and executed separately on single or distributed computing nodes with a decentralized control and supporting mappings to distributed architectures, for instance clusters or clouds. However, the AWARD machine is neutral regarding the scheduling of the workflow activities for execution. For instance, requests for activity execution can be submitted as jobs to an external scheduler.

In the AWARD machine, activities communicate through the AWARD Space abstraction, which supports the links and the corresponding tokens between input and output ports of the workflow activities.

In the following we describe the basic mechanisms of the AWARD Machine that support an operational semantics for executing the AWARD workflows. This will be extended in Chapter 4 in order to describe the characteristics of the AWARD model to support dynamic reconfigurations.

3.4.1 Introduction

In Section 3.3 we described how a workflow developer can specify an AWARD workflow by providing a specification of the workflow activities and their *Tasks*, and by specifying the logical dependencies between activities, expressed through the interconnection of the activity ports.

As illustrated in Figure 3.9 the above declarative specification of an AWARD workflow is interpreted by what we call the AWARD Machine. The implementation of the AWARD Machine abstract architecture and a set of related tools allows the execution of workflows specified according to the AWARD model.

The AWARD Machine performs the operations required to fulfill the semantics of the AWARD model. These operations can be subdivided into two main classes:

1. Operations to execute the life-cycle (iteration steps) for each individual workflow activity, also including the dynamic loading and the invocation of the activity *Task* code;

2. Operations to satisfy the token-driven coordination of all the workflow activities, by performing the required synchronization and the data communication through the connected activity ports.

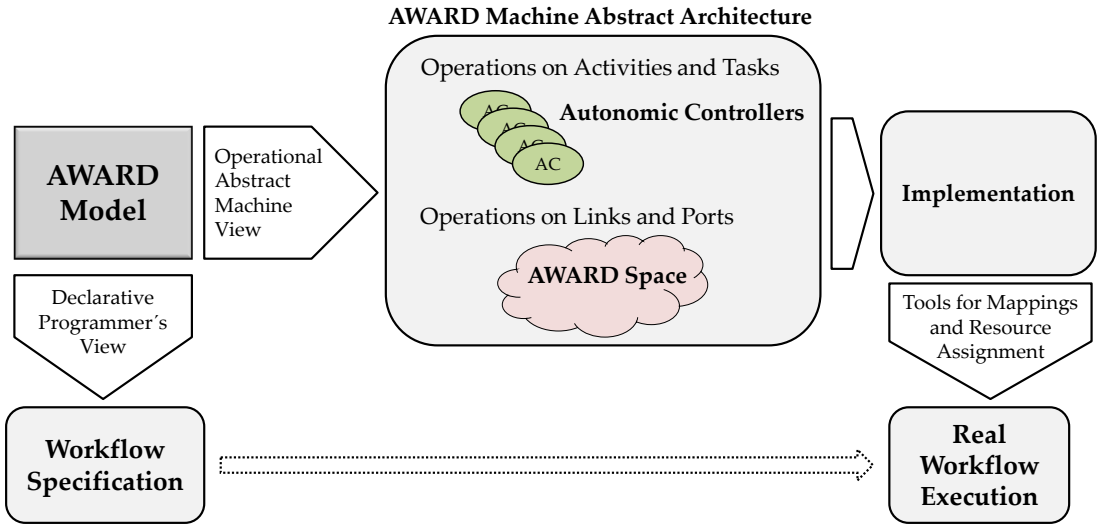


Figure 3.9: The AWARD Machine abstract architecture

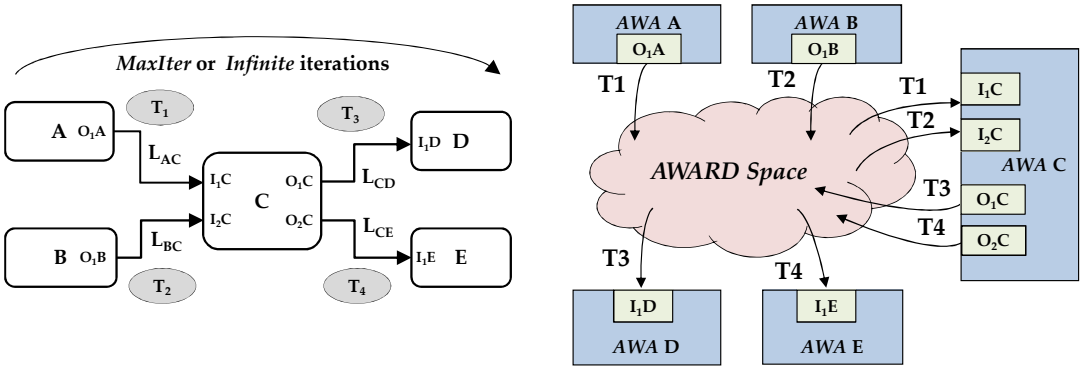
In order to comply with the AWARD design philosophy based on autonomic and decentralized control concepts, the AWARD Machine is designed as a collection of cooperating autonomic controllers, each one responsible for the management of the operations involving each individual workflow activity. As described ahead in this section, the *Autonomic Controller* encapsulates a *State Machine* that controls the iteration steps of the workflow activity. It also manages the workflow activity internal context and the rules and conditions under which dynamic reconfigurations are allowed, with the support of an internal *Rules Engine*.

The cooperation among the autonomic controllers is loosely coupled and is only required to preserve the semantics of token-based communication defined by the AWARD model, regarding the semantics of the operations for sending and receiving tokens through ports.

In order to maintain the goal of a decentralized design, all interactions among the autonomic controllers are indirect, and rely on an intermediate logical abstraction called the AWARD Space. By using this approach, the production of tokens by activity output ports is a write operation into the AWARD Space that only involves the sender activity with a non blocking semantics, as far as the AWARD Space is able to sustain the unbounded assumption for storing tokens until they are retrieved.

Likewise, the operation for getting tokens into the activity input ports only requires a retrieve operation from the AWARD Space, only involving the *Autonomic Controller* of the receiver activity.

Due to the above, the AWARD Space becomes an essential component of the AWARD Machine architecture. As described in Chapter 4 the AWARD Space is also used to support all required activity interactions for the purpose of dynamic reconfigurations management.



(a) A workflow definition

(b) Interactions among components of the AWARD machine

Figure 3.10: The AWARD model of computation

As an example, consider the workflow graph in Figure 3.10(a) to which corresponds a configuration of the AWARD Machine as shown in Figure 3.10(b). This configuration includes five separate instances of autonomic controllers, one dedicated to each workflow activity, denoted by $AWA\ A$, $AWA\ B$, $AWA\ C$, $AWA\ D$, and $AWA\ E$. Additionally it includes a logical instance of the AWARD Space, as the intermediate component for coordination and interaction. We should note that the above configuration only reflects a logical organization of the AWARD Machine, at an abstract machine level, and does not compromise the implementation of the AWARD architecture on physical computing systems, as discussed in Chapter 5.

Figure 3.10(b) also illustrates the indirect interactions of the five autonomic controllers to exchange tokens through the AWARD Space.

According to the semantics of the AWARD model, the following operations define the operational view of the AWARD Machine for executing the workflow of Figure 3.10:

(a) The autonomic controllers of the A, B, C, D and E activities are separately running in parallel;

(b) The A and B activities produce, at their independent paces, respectively, the $T1$ and $T2$ tokens and store them into the AWARD Space;

(c) The C activity is driven by the availability of the $T1$ and $T2$ tokens in the AWARD Space. When both tokens are available, the C activity consumes them from its I_1C and I_2C input ports. These tokens are mapped to the $Task\ Arguments$ and the $Task$ entry point is invoked;

(d) The *Results* of the $Task$ execution are mapped to the O_1C and O_2C output ports, and the corresponding $T3$ and $T4$ tokens are stored into the AWARD Space;

(e) At their independent paces the D and E activities consume, respectively, the $T3$ and $T4$ tokens from the AWARD Space.

3.4.2 The Semantics of Link Operations and the AWARD Space

The AWARD Space supports implicit links between activities.

Each token $T = (\nu T, I, Seq, destInName)$ (see Definition 3.4 on page 58) stored into the AWARD Space has the $destInName$ field to identify the unique name of the destination input port. An AWA activity consumes tokens on its input ports by associative pattern matching where the names of input ports are interpreted as unique keys among all tokens stored into the AWARD Space.

For instance, in Figure 3.10 if the C activity is starting the K^{th} iteration and its I_1C and I_2C input ports are in $Iteration$ mode then the *Autonomic Controller* of the C activity blocks until consuming both tokens $T_1 = (\nu T_1, K, -, I_1C)$ and $T_2 = (\nu T_2, K, -, I_2C)$ where, "$-$" (the field corresponding to the sequence number) denotes "*don't care*".

After the completion of the $Task$ execution and the mapping of results to the O_1C and O_2C output ports, the tokens $T_3 = (\nu T_3, K, Seq, I_1D)$ and $T_4 = (\nu T_4, K, Seq, I_1E)$ are stored into the AWARD Space, thus enabling, respectively, the D and E activities.

The computation model of the AWARD Machine assumes that the AWARD Space supports an unbounded set of tokens for all links of a workflow. This allows AWA activities to produce and to consume tokens at different paces according to their execution speeds in each iteration. Despite this assumption, for a given workflow with a maximum number of iterations, the maximum number of tokens in the AWARD Space is always defined deterministically. For instance, if the workflow of Figure 3.10 has N iterations ($MaxIter = N$), the maximum number of tokens is $N + N$ corresponding to the case where the A and B activities succeed in producing all their tokens before the C activity has consumed any one. However, if a workflow has an infinite number of iterations and the upstream activities produce tokens at a very high pace and the downstream activities consume them at very slow pace, the number of accumulated tokens in the AWARD Space is unbounded. In such extreme situations, due to the physical limits of the AWARD Space, the workflow execution can fail.

The computation model of the AWARD Machine also assumes that links are reliable. This requires an implementation ensuring that there is no token loss during communication and ensuring a persistent storage in the AWARD Space.

In the AWARD model, links are implicitly defined as pairs that associate names of output ports to one or more names of input ports (as in Definition 3.9 on page 61). Links carry tokens marked with their destination input ports (as in Definition 3.4 on page 58). Input and output ports have unique global names (as in Definitions 3.7 and 3.8, respectively on pages 60 and 60).

From an operational view, links are abstractions implicitly represented by sets of tokens emitted anonymously by any output port and marked by the unique global name of the destination input port.

In order to meet the requirements to support the link abstraction and the support to asynchronous communication we defined the abstraction of a global data store space, named the AWARD Space, which is characterized by the following properties:

- **Unbounded size:** The AWARD Space is unbounded for storing all tokens of all links independently of the token production pace by all workflow activities. However, physical limits can impose restrictions upon this property, namely the limit of memory on the computational resources used to support the AWARD Space operation;
- **Associative (Content addressable):** The access to tokens must allow to get tokens by any of their fields or the tokens fields can be indexed as keys for future retrieval;
- **Persistent write:** A non-blocking operation which allows writing tokens. The AWARD Space must ensure the persistence of tokens on non-volatile memory in a transparent way;
- **Atomic retrieval:** A blocking operation for retrieving tokens using any of their fields as keys for pattern matching. This operation must be atomic in the presence of concurrent requests for retrieving a token. On success the operation removes the token from the AWARD Space;
- **Subscribing:** The AWARD Space must allow implementing the observer or the publish/subscribe messaging software design patterns [Gam+95], by maintaining a list of registered subscribers interested in specific tokens. When these tokens are written, the AWARD Space automatically notifies the subscribers. This property is mainly used to provide asynchronous notifications that trigger the event handlers in the autonomic controllers, for instance for managing dynamic reconfigurations.

3.4.3 The Life-cycle of an Autonomic Activity

The operation of the AWARD machine is based on autonomic workflow activities (AWA) that can be separately executed, possibly in distinct computer nodes. The operation of each workflow activity is managed by an *Autonomic Controller* (AC), which controls the life-cycle of each AWA activity, including the control of iterations and the support for dynamic reconfigurations. For each iteration, the *Autonomic Controller* performs the following main actions:

1. *GetTokens:* Getting input tokens from the AWARD Space to all enabled input ports specified in the workflow activity specification;
2. *MapInputs:* Mapping the values of these tokens to an array of *Arguments* used to invoke the activity *Task*;

3. *InvokeTask:* According to the activity specification, the *Autonomic Controller* dynamically loads the software component developed to implement the *Task* and invokes the entry point function, passing the array of *Arguments* and an array of *Parameters* specified in the activity specification;

4. *MapOutputs:* After *Task* completion, the *Autonomic Controller* gets the *Results* returned by *Task* and maps them to output tokens, to be sent by the enabled output ports;

5. *PutTokens:* Put the tokens of the enabled output ports into the AWARD Space. The tokens are marked with the destination input ports;

6. *IterationsControl:* Verify if the current iteration number is the maximum iteration number specified for that activity. If the last iteration was reached then the *Autonomic Controller* terminates the activity, else the *Autonomic Controller* increments the current iteration number and repeats the first action (step 1.) of the life-cycle.

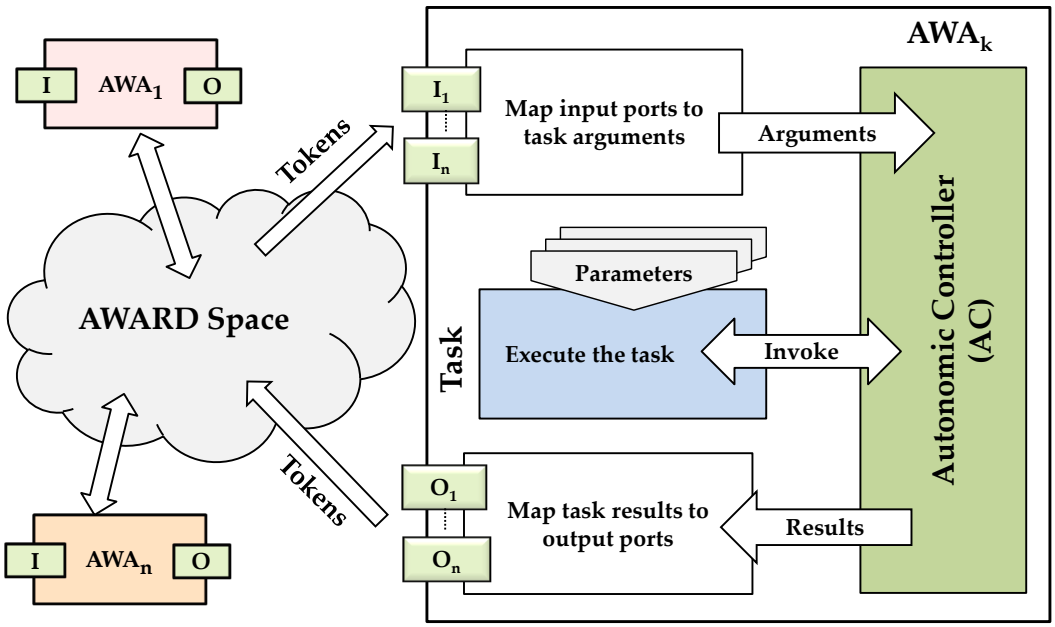


Figure 3.11: Model of an AWA and interactions through the AWARD Space

As shown in Figure 3.11, firstly, the AWA *Autonomic Controller* takes the matching tokens from the AWARD Space for each enabled input port and applies the mappings of tokens to the *Task Arguments*. Secondly, the software component that implements the *Task* is dynamically loaded and its *EntryPoint* is called. Thirdly, the *Results* from *Task* execution are mapped to the enabled output ports of the AWA activity, and the corresponding output tokens are generated and stored into the AWARD Space until they are consumed by destination activities. The AWA *Autonomic Controller* ensures that input and output tokens for each iteration are handled in the AWARD Space as distinct.

Also, the token processing orderings are enforced by identifying tokens with the current iteration number or a sequence number, according to the operation token modes defined in the AWA input and output ports (Definitions 3.7 and 3.8 respectively on pages 60 and 60).

The internal details of the *Autonomic Controller* are described later in this section.

The AWA activities of AWARD workflows are data driven and decentralized, allowing each activity to start, to run and to terminate separately without any centralized execution engine.

During the activity life-cycle it is possible for a user or an external tool to interact asynchronously with each *Autonomic Controller*.

By using the *Subscribing* property of the AWARD Space, each *Autonomic Controller* registers the adequate handlers for subscribing to asynchronous events with multiple purposes: i) getting information on the internal state and context of an activity; ii) explicitly forcing an AWA activity termination; iii) processing the dynamic reconfiguration operators as presented in Chapter 4.

3.4.4 The Semantics of Task Invocation

In order to specify the workflow activities the developer only needs to know the names of the software components used for implementing the activity *Tasks* and specifying their logical dependencies through the input and output ports. This is an important characteristic of the AWARD model as it allows the development of *Task* algorithms by any programmer with expertise on the application problem domain, and eases their integration into workflows.

Nowadays in multiple computational science domains it is common to provide software libraries with algorithms to solve specific problems. The AWARD model supports this strategy by allowing the activity *Tasks* to be developed as object-oriented classes inside specific problem-domain software libraries.

The only programmer's commitment is to develop these classes with an entry-point function (*EntryPoint*) obeying a generic interface defined by the following signature:

public Object[ ] *EntryPoint*(Object[ ] *Arguments*, Object[ ] *Parameters*)

In order to illustrate the semantics of *Task* invocation, consider a software library named *awardlib* and a class with name *SomeTask*, which implements the above generic interface. If this class is used to execute an activity *Task*, the absolute name of the *Task* that is used in the activity specification is *awardlib.SomeTask*.

When the *Autonomic Controller* needs to invoke the activity *Task*, it dynamically loads the class, creates an object instance and calls the method *EntryPoint* by passing the *Arguments* obtained from the activity input ports and the activity *Parameters* as arrays of objects.

We assumed that the type of objects passed as *Parameters* is always the *String* type. The types of objects passed as *Arguments* are types according to the activity input ports specification.

After activating the *Task*, the *Autonomic Controller* waits for *Task* completion. If the *Task* completes with success the *Autonomic Controller* receives an array of objects with the *Results* produced by the *Task*. The types of these objects are types according to the activity output ports specification. However, a *Task* can fail by throwing an exception, which leads the *Autonomic Controller* into a fault state.

3.4.5 The Operation of the Autonomic Controller

The *Autonomic Controller* (AC) operation to control an *Autonomic Workflow Activity* (AWA) is depicted in Figure 3.12. The internal components of the *Autonomic Controller* interact with the AWARD Space for getting and putting tokens from/to the activity input and output ports, and for handling external asynchronous events. The latter are generated by requests performed by *Tools* through an application programming interface named *Dynamic API*.

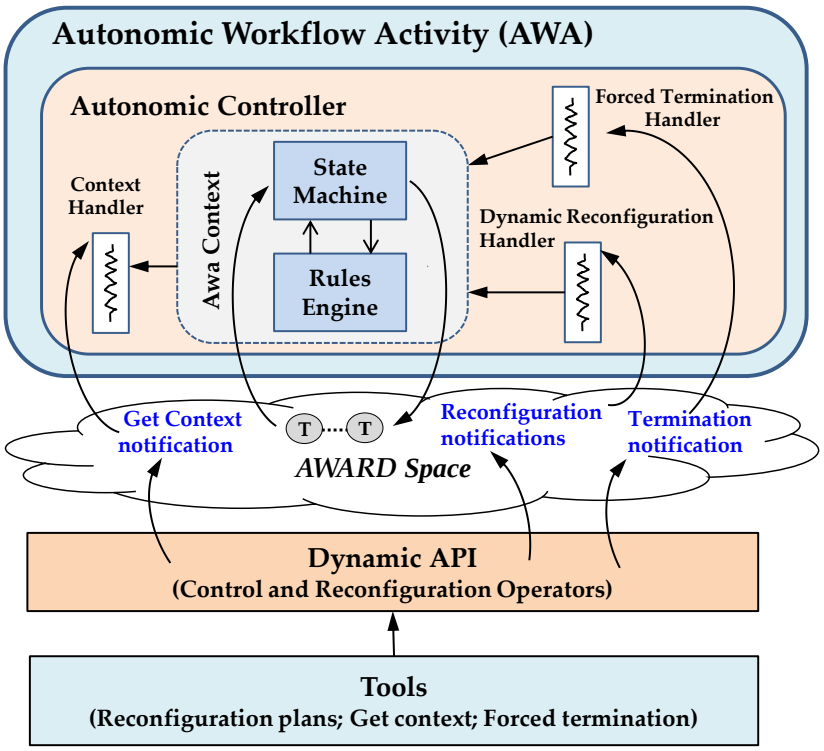


Figure 3.12: The interactions of the *Autonomic Controller* with the AWARD Space

Under the AWARD Space a set of *Tools* generate asynchronous events, for instance requests for dynamic reconfigurations, which are published as notifications into the AWARD Space by using a software library, which provides an application interface (*Dynamic API*).

Tools can also submit notifications for requesting information related to the internal state and context of the activity. Additionally the *Autonomic Controller* can be notified for forcing the termination of the activity.

A detailed description of *Tools* and the *Dynamic API* interface is presented in Chapter 5.

3.4.5.1 The Internal Organization of the Autonomic Controller

As depicted in Figure 3.12 the *Autonomic Controller* is internally composed of the following components:

1. The *AWA Context*, consisting of two types of information. One is the information related to the actual activity configuration, for instance the state of the input and output ports, and the second type of information is related to the internal state variables of the *Autonomic Controller*, for instance the current iteration number;

2. A *Rules Engine*, which supports a knowledge base consisting of a set of facts according to the *AWA Context*. Facts are arbitrary patterns of data records related to the *AWA Context*. The *Rules Engine* also supports a set of rules for acting as *if-then* statements triggered dynamically by conditions based on facts. When rules trigger, the knowledge base is modified by retrieving or adding new facts to change the *AWA Context*;

3. A *State Machine*, which controls the execution steps of the *Autonomic Controller* according to its life-cycle and according to the *AWA Context*. For example, the *State Machine* gets and puts tokens through the AWARD Space, according to the input and output ports configuration, invokes the activity $Task$, and handles the possible faults;

4. A set of event handlers, for subscribing and managing asynchronous events, such as related to requests for dynamic reconfigurations, for explicit AWA termination, and for getting information on the current *AWA Context*. Such requests are published into the AWARD Space by tools through the *Dynamic API* interface activating the corresponding AWA callback handlers, which were previously registered into the AWARD Space.

3.4.5.2 The AWA Context

As shown in Figure 3.12, internally to the *Autonomic Controller* the *State Machine* interacts with a *Rules Engine* in order to maintain two types of information related to the activity specification and to the *Autonomic Controller* operation.

The first information type is related to the configuration of the activity according to the workflow specification, for instance, the definition of the input and output ports, the

Table 3.2: The AWA Context

Context	Information Content	
Configuration	Pars:	The activity *Parameters*;
	Task:	Name of software component that implements the activity *Task*;
	Inputs:	The set of activity input ports;
	Mapins:	The Mapping of activity input ports to *Arguments* of the activity *Task*;
	Outputs:	The set of activity output ports;
	Mapouts:	The Mapping of *Task Results* to the activity output ports;
	MaxIter:	The activity maximum number of iterations.
Internal	IsToStart:	A condition to indicate if the activity starts immediately or needs to wait for a signal according to the tool used for launching an activity
	CurIter:	The number of the current AWA iteration;
	CurState:	The current state of the *State Machine*;
	IsFaulty:	A condition to indicate that a failure has occurred and the *State Machine* is in a *fault* state;
	InsReady:	A condition to indicate that all enabled inputs have received tokens;
	InsTokens:	The list of tokens received from all inputs on the current iteration number;
	OutsTokens:	The list of output tokens sent by the output ports in the current iteration number;
	TArgs:	The list of *Arguments* to invoke the activity *Task* on the current iteration number;
	TRes:	The list of *Results* returned by the activity *Task* on the current iteration number.

activity *Task* name and the maximum number of iterations. Note that this information can be changed by submitting dynamic reconfiguration plans as we discuss in Chapter 4.

The second information type is related to the current operational state of the AWARD machine, for instance, the value of the current iteration number, the current state of the *State Machine* and the current input and output tokens.

The union of these types of information is named the *AWA Context*. In terms of the operational view of the AWARD Machine, the *AWA Context* information, stored as facts into the *Rules Engine*, represents the knowledge data base for the operation of the *Autonomic Controller*.

As presented in Table 3.2, the *AWA Context* consists of two parts. The first part, named *Configuration Context*, is related to the workflow specification and it embodies the information on the current activity configuration. The second part, named *Internal Context*, embodies the information of a set of global variables related to the operation of the *Autonomic Controller* during the AWA execution.

The *AWA Context* of the A activity, denoted by $Ctx(A)$ is defined by the following definitions:

Definition 3.17: *The AWA Context:* $Ctx(A)$

The context of the A activity is the $Ctx(A) = \{CfCtx(A), IntCtx(A)\}$ set that contains the AWA configuration context, denoted by $CfCtx(A)$ and the internal *AWA Context*, denoted by $IntCtx(A)$.

The *Configuration Context*, denoted by $CfCtx(A)$, is related to the basic definitions of an AWA activity, namely the definition of what is an AWA activity (as in Definition 3.5 on page 59), with a set of inputs $A_{Inputs} = \{I_1A,...,I_nA\}$, a set of outputs $A_{Outputs} = \{O_1A,...,O_nA\}$ and the corresponding mappings $AMapins$ (as in Definition 3.13 on page 63) and $AMapouts$ (as in Definition 3.14 on page 63).

Definition 3.18: ***The Configuration context:*** $CfCtx(A)$

The configuration context of the A activity, denoted by $CfCtx(A)$ is defined by the following tuple:
$CfCtx(A) = (APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$
whose fields are:

1. The activity *Parameters*, denoted by $APars$;
2. The name of AWA *Task*, denoted by $ATask$;
3. The configuration context of the AWA inputs denoted by $CfCtx_{inputs}$, defined below in Definition 3.19;
4. The corresponding inputs mapping to the *Task Arguments*, denoted by $AMapins$ (Definition 3.13 on page 63) ;
5. The configuration context of the AWA outputs, denoted by $CfCtx_{outputs}$, defined below in Definition 3.20;
6. The corresponding outputs mapping from *Task Results*, denoted by $AMapouts$ (Definition 3.14 on page 63)
7. The maximum number of iterations, denoted by $MaxIter$.

Definition 3.19: ***The Configuration context of AWA inputs:*** $CfCtx_{inputs}$

The configuration context of AWA inputs, denoted by $CfCtx_{inputs}$, is defined by the following set:
$CfCtx_{inputs} = \{CfCtx_{in}(I_1A),...,CfCtx_{in}(I_nA)\}$
containing the configuration contexts of all input ports where the configuration context of an input port (according to Definition 3.7 on page 60) is defined as:
$CfCtx_{in}(I_iA) = (I_iA, Ttype, IMode, State)$, $1 \leqslant i \leqslant n$.

Definition 3.20: ***The Configuration context of AWA outputs:*** $CfCtx_{outputs}$

The configuration context of AWA outputs, denoted by $CfCtx_{outputs}$, is defined by the following set:
$CfCtx_{outputs} = \{CfCtx_{out}(O_1A), ..., CfCtx_{out}(O_nA)\}$
containing the configuration contexts of all output ports where the configuration context of an output port (according to Definition 3.8 on page 60) is defined as:
$CfCtx_{out}(O_iA) = (O_iA, Ttype, OMode, SendTo, State)$, $1 \leqslant i \leqslant n$.

Definition 3.21: ***The Internal context:*** $IntCtx(A)$

The internal context of the A activity, denoted by $IntCtx(A)$, is defined by the following tuple:

$$IntCtx(A) = (IsToStart, CurIter, CurState, IsFaulty, InsReady, InsTokens, OutsTokens, TArgs, TRes)$$

containing the fields presented in the second part, (*Internal*), of the Table 3.2 on page 77. Each field is an internal global variable of the *Autonomic Controller*.

3.4.5.3 The State Machine and the Rules Engine

Each *Autonomic Controller* controls the life-cycle and behavior of its associated AWA activity taking the advantage of joining a *Rules Engine* to a *State Machine*. The *Rules Engine* supports a predefined initial set of facts and rules used by the *State Machine* that can be distinctly specified for each workflow AWA activity. As presented in Chapter 4, Section 4.3.2, the *State Machine* is extended to include the support for dynamic reconfigurations. Therefore in this section we describe the *State Machine* without considering the states related to dynamic reconfigurations.

As depicted in Figure 3.13 the *State Machine* has a set of states for performing the operation of the *Autonomic Controller* according to the following phases:

- **Starting:** The *start* and *init* states for initializing the *Autonomic Controller*. Typically the entry point state of the *State Machine* is the *start* state where the *Autonomic Controller* waits for a synchronization signal issued by a tool used to launch the workflow activities. The *init* state initializes the *AWA Context* and begins the immediate operation of the *Autonomic Controller* on the executing phase;

- **Executing:** The main states are repeated over the multiple iterations for executing the main actions of the *Autonomic Controller*. The *idle* state controls the current iteration number to check if the maximum iteration number has been reached and consequently terminating the execution of the activity. The *input* state gets the tokens for all enabled input ports from the AWARD Space. The *mapin* state maps

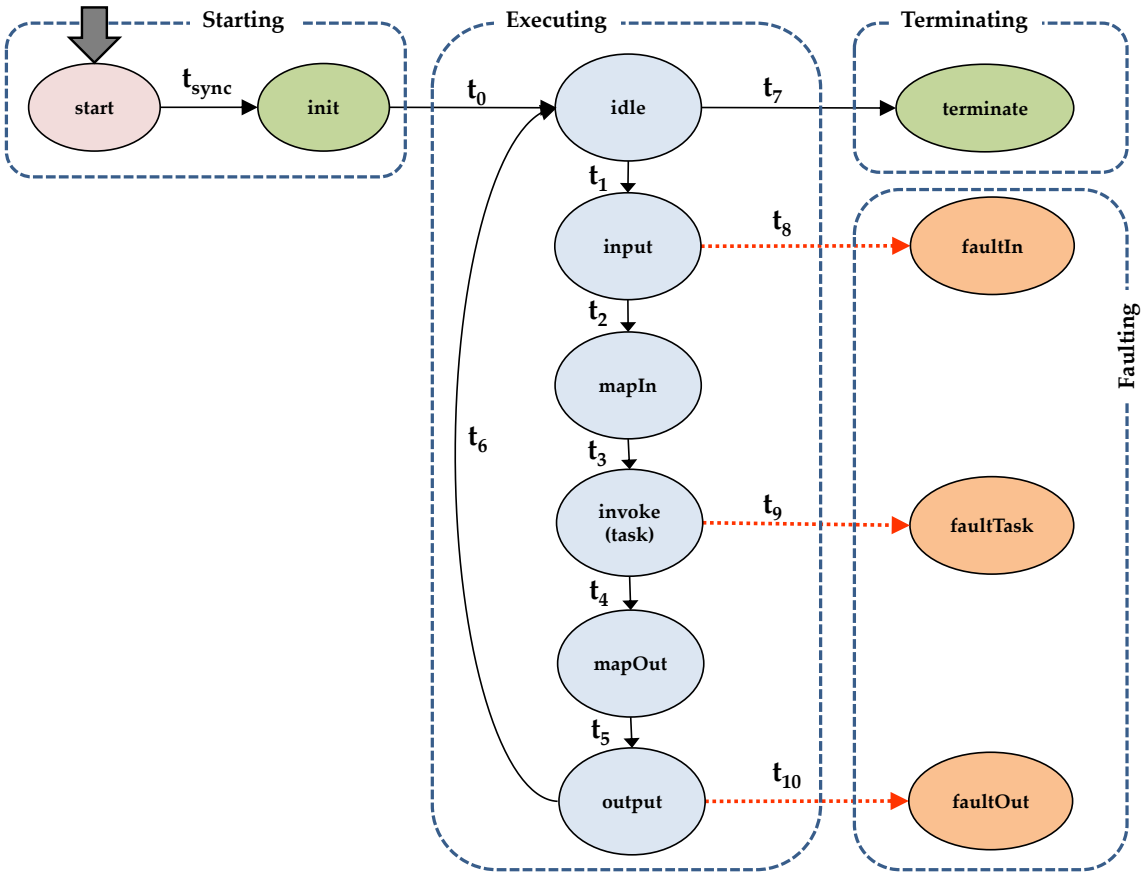


Figure 3.13: The states and transitions of the *State Machine*

the token values to the *Arguments* array of the activity *Task*. The *invoke* state calls the entry point of the software component that implements the *Task* using the mapped *Arguments* and the activity *Parameters*. The *mapout* state maps the array of *Results* returned by the *Task* to token values to be sent through the activity output ports. The *output* state puts the tokens for all enabled activity output ports into the AWARD Space;

- **Terminating:** The *terminate* state defines the end of execution of the *Autonomic Controller* and the completion of the workflow activity;

- **Faulting:** During execution, the *Autonomic Controller* interacts with external resources, where failures can occur. In *input* and *output* states, accessing the AWARD Space can fail due to communication problems, or when the physical limits of the storage capacity are reached. In the *invoke* state the software component that implements the activity *Task*, can also fail. Therefore the *faultIn*, *faultTask* and *faultOut* states are reached when failures happen, respectively, in states *input*, *invoke* and *output*. On these failure states, the *Autonomic Controller* saves the *AWA Context* as logging information to facilitate debugging and failure recovery. In Chapter 4 we

illustrate how failure recovery can be achieved by applying dynamic reconfigurations.

The association between the *State Machine* and the *Rules Engine* provides flexibility in the operation of the *Autonomic Controller*, for example to control the *State Machine* transitions. As depicted in Figure 3.13 the entry point state of the *State Machine* is the *start* state where the *Autonomic Controller* waits for a synchronization signal. This entry point state is specified as a default fact of the *Rules Engine*. However, the workflow developer can change this fact in order to specify that the entry point of the *State Machine* is the *init* state, in order to override the need for a starting synchronization signal. Also, as presented ahead in this section, the state transitions are controlled by firing rules in the *Rules Engine*.

✧ **Definitions related to the *State Machine***

In each state of the *State Machine*, the *Autonomic Controller* performs the following steps:

1. Execution of the corresponding state actions (a_s);
2. Evaluation of conditions (C) in order to decide on the state transition to the next state;
3. Execution of actions (a_t) associated to the state transition.

Definition 3.22: *The State Machine:* SM

The *State Machine* is defined as the $SM = (S, T, V, C, A)$ tuple, consisting of a set of states (S), a set of state transitions (T), a set of global variables (V), a set of conditions (C), and a set of actions (A) that can be actions performed in the state (a_s), or actions performed during state transitions (a_t).

Definition 3.23: *The State Machine states:* S

The set of states (S) is defined as,

$$S = \{start, init, idle, input, mapIn, invoke, mapOut, output, terminate, faultIn, faultTask, faultOut\}$$

and the corresponding actions (a_s) are described in Table 3.3.

✧ **Transitions**

Definition 3.24: *A State Machine transition:* t

A state transition denoted by t is defined as a tuple $\forall t \in T,\ t = (s_i, s_j, c, a_t)$ where (s_i) is the source state and (s_j) is the destination state, (c) is a condition that enables the state transition and (a_t) is a set of actions performed on the state transition. When a transition has no actions the action set (a_t) is denoted by a dash (−).

Table 3.3: The states and actions of the *State Machine*

State	Description	Actions of the *Autonomic Controller* (a_s)
start	The entry point of the *State Machine* to wait for a start synchronization signal issued by the tool that launched the activity	It waits for a start synchronization signal then it puts the global variable $IsToStart$ to TRUE and performs the t_{sync} transition.
init	The initialization state of the *State Machine* used to initialize the internal *AWA Context*.	It creates facts into the *Rule Engine* according to the *AWA Context* information (Definitions 3.17, 3.18, and 3.21) and performs the t_0 transition.
idle	The state to control the number of iterations in order to verify if the last iteration ($MaxIter$) is or not reached	It performs the t_7 transition if the last iteration is reached or the t_1 transition to begin a new iteration.
input	The state for getting tokens from the AWARD Space for all inputs of the activity by ensuring that each input port has a token according to its configuration.	It gets the tokens from the AWARD Space for all activity input ports and performs the t_2 transition; otherwise if the access to the AWARD Space causes failures, the t_8 transition is performed leading to the *faultIn* state.
mapIn	The state for mapping the token values received on the *input* state to an array of *Arguments* (Definition 3.12) used in the activity $Task$ invocation.	It executes the input mapping function and performs the t_3 transition.
invoke	The software component that implements the AWA $Task$ ($ATask$) is loaded, instantiated and the entry point function ($EntryPoint$) is called using the array of arguments ($Arguments$) prepared in the *mapIn* state (Definition 3.12) and the configured *Parameters* ($APars$).	It invokes the activity $Task$ and performs the t_4 transition for mapping the array of *Results* (Definition 3.12) to the output ports, otherwise if any failure occurs during the $Task$ invocation the t_9 transition is performed leading to the *faultTask* state.
mapOut	The state for mapping the array of *Results* (Definition 3.12) returned from the $Task$ to token values to be sent in the *output* state.	It executes the output mapping function and performs the t_5 transition.
output	The token values mapped in the *mapOut* state are associated to tokens according to the configuration of the activity output ports. These tokens are written in the AWARD Space.	It puts the output ports tokens in the AWARD Space and loops back by performing the t_6 transition to the *idle* state to start the next iteration, otherwise if the access to the AWARD Space causes failures the t_{10} transition is performed leading to the *faultOut* state.
terminate	This state defines the end of the life-cycle of an activity.	It performs clean-up actions. For instance unregister event handlers and close the connection to the AWARD Space.
faultIn *faultTask* *faultOut*	The states to wait for possible recovery actions based on dynamic reconfigurations. (see Section 4.3.2 on page 113)	It reports the *AWA Context* and the reasons for the failure as logging information.

Definition 3.25: *The State Machine transition set:* T

The set of state transition (T) is defined as,

$$T = \{t_{sync}, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$$

and each transition tuple (t), (Definition 3.24), is detailed in Table 3.4 on page 83.

✧ **Actions on state transitions**

As presented in Table 3.4 some transitions have actions to be performed during the state transition. The action performed in t_6 corresponds to increment the current iteration number $a_6 = (CurIter = CurIter + 1)$. The actions $a_8, a_9, a_{10} = SaveContext$ performed in the t_8, t_9, t_{10} transitions, correspond to saving the information of the current *AWA Context* to be logged in the faulty states for facilitating debugging and failure handling. The $SaveContext$ action is discussed in Chapter 5, which details the *Autonomic Controller* implementation.

Table 3.4: The state transitions

Transition (t)	**Source State** (s_i)	**Destination State** (s_j)	**Condition** (c)	**Action** (a_t)
t_{sync}	*start*	*init*	c_{sync}: The variable $IsToStart$ is equal to TRUE indicating that the activity must start immediately.	–
t_0	*init*	*idle*	c_0: Always TRUE	–
t_1	*idle*	*input*	c_1: The maximum iteration number has not been reached.	–
t_2	*input*	*mapIn*	c_2: All enabled inputs have tokens and there are no failures in the access to the AWARD Space.	–
t_3	*mapIn*	*invoke*	c_3: Always TRUE.	–
t_4	*invoke*	*mapOut*	c_4: There are no failures on $Task$ invocation.	–
t_5	*mapOut*	*output*	c_5: Always TRUE.	–
t_6	*output*	*idle*	c_6: There are no failures in the access to the AWARD Space.	a_6: Increments the current iteration.
t_7	*idle*	*terminate*	c_7: The current iteration reaches the maximum iteration number.	–
t_8	*input*	*faultIn*	c_8: A failure occurs when accessing the AWARD Space for getting the tokens of the input ports.	a_8: Saves the current internal *AWA Context* ($SaveContext$).
t_9	*invoke*	*faulTask*	c_9: A failure occurs when invoking the $Task$.	a_9: Saves the current internal *AWA Context* ($SaveContext$).
t_{10}	*output*	*faultOut*	c_{10}: A failure occurs when accessing the AWARD Space for putting the tokens on the output ports.	a_8: Saves the current internal *AWA Context* ($SaveContext$).

✧ **Global variables of the State Machine**

Definition 3.26: ***The State Machine global variables:*** V

The *State Machine* has a set of global variables V defined as,

$$V = \{IsToStart, CurIter, CurState, IsFaulty, InsReady, InsTokens, OutsTokens, TArgs, TRes\}.$$

These variables define the internal *AWA Context* (Definition 3.21 on page 79).

✧ **Conditions**

Definition 3.27: *Conditions:* C

The condition set $C = \{c_{sync}, c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$ defines the conditions associated to the state transitions. These conditions are TRUE or FALSE depending on the evaluation of the expressions involving the global variables as presented in Table 3.5.

Table 3.5: The conditions for the state transitions

$c_{sync} = (IsToStart == TRUE)$	$c_4 = not\ IsFaulty$
$c_0 = TRUE$	$c_5 = TRUE$
$c_1 = (CurIter < MaxIter)$	$c_6 = not\ IsFaulty$
$c_2 = (not\ IsFaulty\ \&\ InsTokens)$	$c_7 = (CurIter == MaxIter)$
$c_3 = TRUE$	$c_8, c_9, c_{10} = IsFaulty$

✧ **The Rules Engine**

The global variables (V) and conditions (C) of the *State Machine* are managed using the *Rules Engine*. Global variables and internal variables of the *State Machine* are stored into the *Rules Engine* as facts denoted by tuples with the $(variable\ value_1, ..., value_n)$ pattern. Rules act as *if-then* statements triggered by conditions based on facts.

As an example, in Listing 3.2 we present the pseudo-code of a rule for triggering the state transitions of the *State Machine*. During the lifetime of the *Autonomic Controller* there is always the fact *(CurState value)* for containing the value of the $CurState$ global variable. When the fact *(ChangeState value next)* is inserted to change the state value to the state *next*, the rule *ChangeStateMachineState* fires and inserts a fact to represent the new value of the global variable *(CurState next)*. The facts that caused the rule firing are deleted to avoid a retrigger of rules according to the common semantics of the *Rules Engine*.

Listing 3.2: Example of a rule for allowing change the state of the *State Machine*

```
1  BeginRule ChangeStateMachineState
2  IF
3     ExistFact f1=(CurState value)
4  AND
5     ExistFact f2=(ChangeState value next)
6  THEN
7     DeleteFact f1
8     DeleteFact f2
9     InsertFact (CurState next)
10 EndRule
```

As another example when a failure occurs, a fact with the pattern *(Failure reason)* is inserted into the working memory of the *Rules Engine* to indicate the failure and the corresponding reason. In Listing 3.3 the pseudo-code of a rule is presented to force the *State Machine* to enter the $faultTask$ state when a failure occurs during the $Task$

Listing 3.3: Example of a rule to force the *faultTask* state of the *State Machine*

```
1 BeginRule  Failure:
2 IF
3    ExistFact f1=(CurState invoke)
4 AND
5    ExistFact f2=(Failure reason)
6 THEN
7    InsertFact (ChangeState  invoke  faultTask)
8 EndRule
```

invocation. When this rule fires, the fact *(ChangeState invoke faultTask)* is inserted into the working memory, to fire the previous rule (Listing 3.2) in order to lead the *State Machine* to the $faultTask$ state.

3.4.5.4 The Workflow Termination

The semantics of the workflow termination depends on the workflow specification and the conditions that can occur during the workflow execution as illustrated in Figure 3.14.

According to the AWARD model, usually a workflow has a maximum number of iterations ($MaxIter$) such that all workflow activities execute the same number of iterations. Due to the autonomic characteristics of an activity the point where it reaches its maximum number of iterations depends on its own processing pace.

The implicit termination of an activity occurs when it reaches the maximum number of iterations ($MaxIter$) and the *State Machine* reaches the *terminate* state. Therefore the execution of a workflow instance terminates orderly when all of its activities have already reached the *terminate* state.

When a workflow terminates orderly, and all tokens produced by its activities were consumed then the AWARD Space is empty, that is, there are no application tokens left.

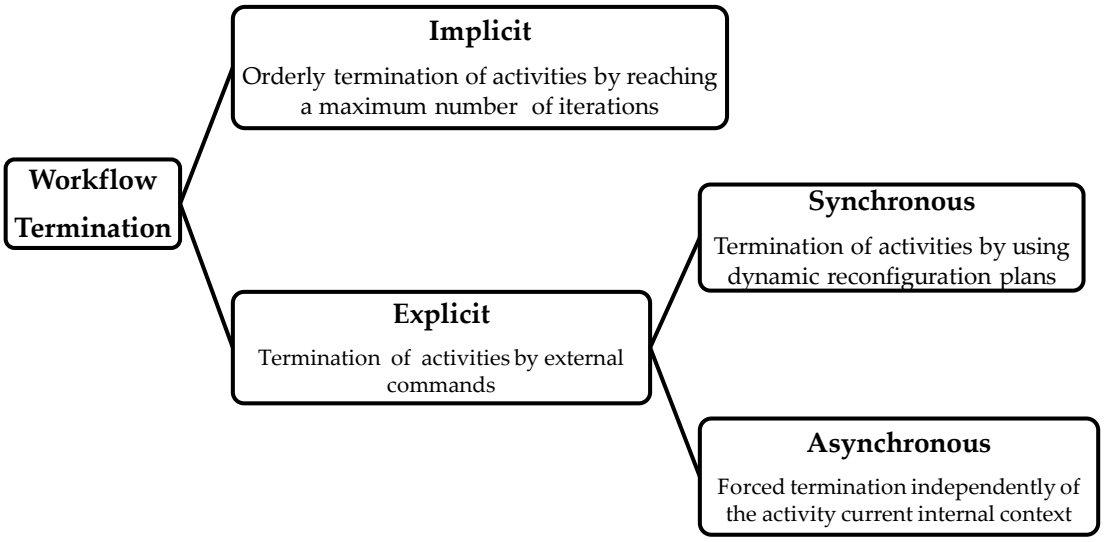


Figure 3.14: The semantics of the workflow termination

However, some workflow scenarios introduce a requirement for explicit termination of activities. The AWARD model supports synchronous explicit termination of activities by applying dynamic reconfiguration plans (Chapter 4, Section 4.2 on page 101).

For example, when a workflow was initially specified with an infinite number of iterations, the termination of activities can be performed by explicitly applying a dynamic reconfiguration plan to redefine a finite maximum number of iterations equal for all activities. In this case the semantics of the workflow termination falls into the situation where all activities have already reached the *terminate* state.

Another possibility is to submit dynamic reconfiguration plans with a command to terminate separately each workflow activity. However, the success of invoking a dynamic reconfiguration plan with a command to terminate one activity depends on the internal context of the activity. In fact, the reconfiguration actions for activity termination can never complete if the *State Machine* of the activity is in the *input* state, waiting for tokens that will not be produced anymore by an activity that had terminated first.

In order to ensure the completion of the workflow termination the AWARD machine supports an asynchronous mechanism for forcing the termination of the activities independently of their current internal contexts. This increases the flexibility of the AWARD model by allowing a distinct maximum number of iterations for some activities by initial specification, or as a consequence of applying dynamic reconfiguration scenarios. However, the asynchronous forced termination of activities may leave tokens unconsumed in the AWARD Space.

3.4.5.5 Recovery from Failures

During the execution of the workflow, any *Autonomic Controller* can independently fail on critical situations related to interactions with external entities. Namely, despite the assumption that links between activities are reliable and there is no loss of tokens, an *Autonomic Controller* can lose the connectivity to the external computing resources that implement the AWARD Space, when trying to read or write tokens. Furthermore, an activity *Task* implementation can generate failures caused by algorithm errors or by possible unavailability of the external resources used.

Faults can occur in *input* or *output* states related to the connectivity to the external computing resources of the AWARD Space, when respectively trying to read and write tokens.

The *invoke* state is a critical state where faults can occur. In fact any fault produced within the *Task* code, developed by application developers, such as unreachable resources and network failures, etc., will force the *State Machine* to the *faultTask* state.

When a fault is caught, the *State Machine* reports the internal *AWA Context*, including the failure reasons as facts into the *Rules Engine* and goes to the fault states where, if it is possible, the failure reasons are logged into the AWARD Space. By monitoring the AWARD Space, the workflow developer can try to perform actions based on dynamic

reconfigurations for recovering from the failures or, at worst case, when the failures are persistent the workflow developer can terminate the activity by explicit execution of a forced termination command, which possibly implies aborting the workflow execution.

As presented in Chapter 4 and Chapter 6, the AWARD model supports the recovery from faults by using dynamic reconfigurations. For instance, if the activity *Task* invokes external resources that are unavailable, it is possible to recover by changing the *Task Parameters*, or by changing the complete software component that implements the *Task* in order to redirect the invocation to alternative resources.

However, it is important to note that it is not always possible to recover from faults. As an example, if some actions inside an activity *Task* are not idempotent it may not be possible to reinvoke the *Task* with the same *Arguments* or the same *Parameters*. Another example where the recovery may not be possible is when persistent failures occur during the access to the AWARD Space. For instance, after relaunching or reconnecting to the AWARD Space, it may not be possible to roll back the actions for getting or putting tokens into the AWARD Space.

3.4.5.6 The Event Handlers

Internally, the *Autonomic Controller* of each AWA activity has three event handlers, for subscribing and managing asynchronous events:

1. The *Context Handler* handles requests for getting information related to the current *AWA Context*;

2. The *Forced Termination Handler* handles the request for explicit AWA activity termination;

3. The *Dynamic Reconfiguration Handler* handles requests for performing dynamic AWA activity reconfigurations.

These AWA notification handlers, which were previously registered into the AWARD Space, handle and manage the requested events published into the AWARD Space by tools through a *Dynamic API* interface, whose implementation is detailed in Chapter 5.

For example, the request for obtaining the *AWA Context* can be submitted by any application tool by invoking the function *GetAwaContext*(*AwaName*) of the *Dynamic API* interface, which returns the current internal *AWA Context*.

The AWARD tool *KillAwa*(*AwaName*) ensures the termination of an AWA activity independently of its internal context. The tool submits a request to be handled by the *Forced Termination Handler*.

The requests for AWA activity dynamic reconfigurations are submitted by scripts using a set of operators provided through the *Dynamic API* interface as presented in Chapter 4.

3.5 Chapter Conclusions

In this chapter we presented the AWARD model according to two perspectives. Firstly, we described the fundamental concepts and essential definitions for allowing a workflow developer to specify the AWARD workflows, completely decoupled from the low-level details of the execution infrastructure. Secondly, we presented the operational view of the AWARD Machine abstract architecture for supporting the execution of the AWARD workflows, relying on an *Autonomic Controller* for controlling the life-cycle of the workflow activities and an abstraction called AWARD Space as an unbounded and reliable global data store.

As presented in this chapter, the AWARD model for supporting the execution of scientific workflows has the following main characteristics:

- Ease of specification of AWARD workflows, decoupled from low-level details;
- Autonomic workflow activities (AWA) executable without a centralized control;
- Links between AWA activities are abstractions supported by the AWARD space as a unbounded and reliable global data store;
- The development of the *Tasks* of the AWA activities is decoupled from the underlying execution infrastructure. This allows the workflow developer to focus on the problem domain and not on low-level details related to the implementation of the workflow execution engine.

CHAPTER 4

Dynamic Reconfigurations

The support of the AWARD model for dynamic reconfigurations of long-running workflows. The reconfiguration plans based on a set of basic dynamic reconfiguration operators can be applied to dynamically change the structure and behavior of the AWARD workflows.

This chapter presents an important component of the AWARD model concerning the support for dynamic reconfigurations of long-running workflows where each activity executes a large number or even an infinite number of iterations.

Performing dynamic workflow reconfigurations is required and useful in many application scenarios where the workflow structure and/or the behavior of activities must be changed during workflow execution. As illustrative examples of structural changes we can introduce new activities for data filtering, data monitoring and load balancing. This also includes dynamically introducing feedback loops where a new activity receives intermediate tokens as input and produces feedback tokens to previous activities in the structure. As examples of changing the activity behavior we can change activity *Parameters* or change its *Task*. This has the advantage of allowing the runtime modification of the application algorithms, for example, to facilitate computational steering by multiple users and the handling of *Task* fault recovery.

This chapter is organized as follows. In Section 4.1 we present the rationale for dynamic reconfigurations and the main related concepts. Section 4.2 describes the workflow programmer's view for specifying an AWARD dynamic reconfiguration plan based on a set of basic dynamic reconfiguration operators (Section 4.2.1). A concrete example is presented (Section 4.2.2) to illustrate how the workflow programmer can establish a reconfiguration plan to change the structure and behavior of a long-running workflow.

Section 4.3 reviews the operational view of the AWARD machine with the extensions to the *State Machine* of the *Autonomic Controller* for supporting dynamic reconfigurations. In order to support useful application cases, in Section 4.4 we present a set of dynamic reconfiguration scenarios applied to workflow templates. Section 4.5 concludes the chapter by emphasizing the distinct characteristics of the AWARD model concerning the support for dynamic workflow reconfigurations.

4.1 The Rationale for Dynamic Reconfigurations

The workflow paradigm is a useful approach for developing scientific applications based on the composition of multiple activities, allowing each activity to be reused in different application scenarios, for example, long-running experiments with multiple, possibly infinite numbers of iterations for processing large data sets. These experiments are often dynamic and subject to constantly changing requirements. For example, changes to the goals of the experiments, changes to the activity algorithms in order to modify their functionalities or optimize their performance, changes to the application parameters allowing parameter sweep or to handle events arising during the workflow execution, for instance related to failures. Furthermore, computational steering is an important requirement for the emerging scenarios where transnational research teams are spanning workflows over multiple autonomous organizations. Traditional workflow systems, based on a single centralized workflow management engine, may be unsuitable to support these scenarios.

Whereas the above requirements are not known at workflow design time, sometimes there is the need to stop the entire workflow execution for redesigning the workflow and restarting its execution anew. However, stopping and restarting long-running workflows can lead to a waste of elapsed execution time which may also result in increased costs regarding the used allocated resources. These costs can be significant, for instance, when a public cloud infrastructure is used to execute the workflows.

Therefore, for enabling and accelerating collaborative scientific experiments accompanied by continuous adaptation and improvement it is indispensable to develop workflow systems with support for dynamic workflow reconfigurations and user-driven steering [Dee07; Gil+07].

As presented in Section 2.4.3, in spite of the extensive work related to dynamic reconfiguration models and approaches in multiple contexts, such as distributed systems, operating systems, software engineering architectures or business workflows, unfortunately the most widely used and available scientific workflow systems do not provide sufficient flexibility to support dynamic changes of long-running workflows. For example, the support for dynamically adding new workflow activities is an issue not fully addressed by most of the existing scientific workflow systems. Since the beginning of our research we envisioned a workflow model to accommodate dynamic workflow adaptations and structural changes during the workflow execution. This is motivated by the

need to enable dynamic changes on the structure and behavior of a workflow triggered by distinct events, such as: i) User requests in interactive workflow scenarios; ii) Application tools requests during the workflow execution; and iii) Requests originated from events in the execution infrastructure, such as failures related to unavailability of resources or variations in the quality of service. In general, an effective recovery strategy based on dynamic reconfiguration is a critical issue to enable the efficient execution of long-running workflows by allowing substantial savings in elapsed execution time.

✧ **What is an AWARD dynamic reconfiguration**

The AWARD approach to support dynamic reconfigurations aims at providing flexibility for allowing both structural and behavioral changes during the execution of long-running workflows.

A structural change is any change that involves modifying the topology of the workflow graph, that is, adding or removing activities and adding, removing or changing the links between activities. A behavioral change is any change that involves modifying characteristics of the activities, that is, changing the activity *Parameters*, changing the activity *Task*, or changing properties of the activity input and output ports.

The current workflow configuration corresponds to the set of all contexts of the workflow activities where the context of each activity, $Ctx(A) = \{CfCtx(A), IntCtx(A)\}$ (Definition 3.17 on page 77) is composed of the activity configuration context $CfCtx(A)$ (Definition 3.18 on page 78) and the activity internal context $IntCtx(A)$ (Definition 3.21 on page 79).

In general the configuration context of the workflow activities is initially defined by the workflow specification and during the workflow execution only the internal contexts of the workflow activities change according to the life-cycle of their autonomic controllers. However, during the workflow execution the initial configuration context of the workflow activities can be changed leading to changes in workflow configuration by requests from the users, or application tools, or by events originated by the infrastructure environment. In this dissertation these workflow configuration changes are called *dynamic workflow reconfigurations*.

✧ **What is a dynamic reconfiguration plan**

When a workflow is running, dynamic workflow reconfigurations can be performed successively in different stages of the workflow execution by applying a series of reconfiguration plans.

As depicted in Figure 4.1 during the elapsed execution time (T) the workflow execution starts with an initial configuration (Wcf_0) and ends with a final configuration (Wcf_N) by applying multiple reconfiguration plans (Rcf_0), (Rcf_1),...,(Rcf_i),...,(Rcf_{N-1}).

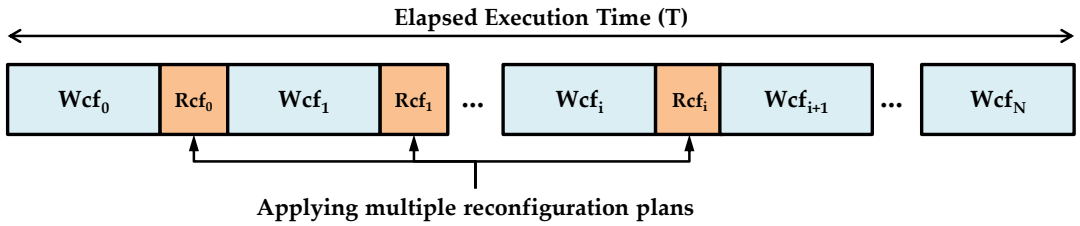


Figure 4.1: Workflow execution with multiple configurations

Definition 4.1: *Workflow configuration:* Wcf_i

A configuration (Wcf_i) of the $W = (Wname, gIterations, G)$ workflow is a set of contexts of all workflow activities A_j in G, with $1 \leqslant j \leqslant n$, that is, $Wcf_i = \{Ctx(A_1), Ctx(A_2), ..., Ctx(A_n)\}$ where the context of the A_j activity, $Ctx(A_j) = \{CfCtx(A_j), IntCtx(A_j)\}$ (Definition 3.17 on page 77) is composed of the activity configuration context $CfCtx(A_j)$ (Definition 3.18 on page 78) and the activity internal context $IntCtx(A_j)$ (Definition 3.21 on page 79).

If there are no dynamic reconfigurations, each workflow activity continues its execution individually according to the semantics of the AWARD computation model and according to dependencies between activities as established through the token-based communication.

In the presence of a dynamic reconfiguration, each activity affected by the reconfiguration plan is subject to the necessary modifications as soon as an adequate execution point is reached. Therefore, a dynamic reconfiguration for an individual A_i activity corresponds to a transition (T) to apply the reconfiguration actions to the current context $Ctx(A_i)_{before}$ of the A_i activity, reaching a new activity context $Ctx(A_i)_{after}$ as depicted in Figure 4.2.

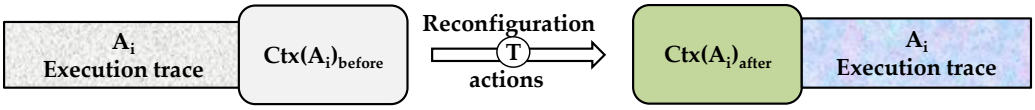


Figure 4.2: Change the activity context by applying the reconfiguration actions

Definition 4.2: *Reconfiguration plan:* Rcf_i

A reconfiguration plan (Rcf_i) is a sequence of changes applied during a workflow execution for modifying the current workflow configuration (Wcf_i) to a new workflow configuration (Wcf_{i+1}) and is defined as a transition $Wcf_i \xrightarrow{Rcf_i} Wcf_{i+1}$. The reconfiguration plan is atomic in the sense that all or none of the changes in the sequence are performed.

A reconfiguration plan (Rcf_i) can modify the current workflow configuration (Wcf_i) to affect the structure of the workflow by changing the links between input and output ports, or by launching new activities. The workflow configuration behavior can also be affected by changing the *Task* and *Parameters* of the workflow activities.

✧ **The scope of a reconfiguration plan**

A reconfiguration plan may not affect the entire workflow but just a workflow subgraph containing a set of activities to be modified.

Definition 4.3: ***The scope of a reconfiguration plan***

The scope of a reconfiguration plan is the set of activities, including possibly new activities, affected by the reconfiguration plan. Each activity A in this set is affected individually by modifications to its configuration context, ($CfCtx(A)$) with changes to its *Parameters* (*APars*), the *Task* name (*ATask*), the input and output contexts ($CfCtx_{inputs}$ and $CfCtx_{inputs}$), the corresponding mappings to *Task Arguments* (*AMapins*) and *Task Results* (*AMapouts*)) and the maximum number of iterations (*MaxIter*). Also, the internal context of the activity ($IntCtx(A)$) can be modified by reconfiguration operators.

✧ **The AWARD approach for supporting dynamic reconfigurations**

According to application requirements or utilization scenarios workflow developers need a high degree of flexibility for dynamically modifying long-running workflows. The AWARD model aims at providing an adequate degree of flexibility to support multiple reconfiguration plans during the workflow execution in order to satisfy the application-dependent change requirements to meet the goals of the application developers.

However, this flexibility regarding the freedom of specifying dynamic reconfiguration plans introduces an essential question related to how to ensure the consistency of the dynamic reconfigurations applied during the execution of long-running workflows.

✧ **Difficulties and related work**

The AWARD model allows the asynchronous execution of activities and also supports multiple users separately launching and controlling autonomic workflow activities on distributed infrastructures. This introduces difficulties related to the coordination of the distinct views of the workflow execution state that are observed by the different participants/users. In fact, each individual user can observe distinct behaviors of the workflow execution, which leads to a large diversity of situations for demanding dynamic reconfigurations of the long-running workflows.

Several works [SLI08; TCBR11] have addressed the issue of ensuring workflow consistency in the presence of dynamic changes, namely related to task rescheduling [CD12], or for handling exception failures [TC+10]. In order to manage changes in business workflows the consistency of modifications to running workflow instances has been addressed in [AC03b; RB07; RRD04]. However, there is not a common and generic definition of

consistency because it depends on the application context, the underlying workflow execution engine and the corresponding execution infrastructure.

Regarding consistency a widely used concept is the workflow soundness [Aal+11] for defining a set of properties as correctness criteria that should be ensured before and after workflow modifications, for instance by ensuring workflow termination, absence of deadlocks and link consistency. Typically these soundness properties are verified by a correctness tool before the workflow execution starts and are afterwards validated when some workflow changes are performed.

✧ **The soundness of a given AWARD workflow configuration**

The semantics of the AWARD model defines the following soundness properties that must be satisfied in any workflow:

Definition 4.4: ***Soundness properties***

An AWARD workflow configuration is sound if the following properties are preserved:

1. *Implicit termination:* All workflow activities terminate after reaching their maximum number of iterations ($MaxIter$);
2. *Input ports reachability:* Given any enabled input port there is at least one enabled output port which produces tokens to be consumed by that input port;
3. *Token delivery:* For any token produced there is one and only one destination input port to consume the token.

Assuming that all links are reliable and unbounded (Section 3.4.2), the properties of the above Definition 4.4, ensure that there is no loss of tokens and all tokens produced are consumed.

✧ **Consistency of an AWARD dynamic reconfiguration plan**

Definition 4.5: ***Reconfiguration plan consistency***

An AWARD dynamic reconfiguration plan $Wcf_i \xrightarrow{Rcf_i} Wcf_{i+1}$ is consistent if Wcf_i and Wcf_{i+1} satisfy the soundness properties of Definition 4.4.

In the following we discuss the AWARD approach concerning the above consistency definition (Definition 4.5). When using the AWARD model, workflow developers can establish workflow reconfiguration plans by relying on a basic set of AWARD operators for performing dynamic modifications to long-running workflows. These reconfiguration plans can involve a single activity or a workflow partition consisting of multiple activities contained in the scope of the reconfiguration plan (Definition 4.3 on page 93).

If the activities involved in the reconfiguration plan are independent from each other,

then after receiving the request to apply the reconfiguration operators, each activity *Autonomic Controller* could separately modify its individual context and continue its execution. However, on one hand the reconfiguration plans can involve multiple activities, which are interdependent through the production and consumption of tokens. On the other hand, the production and consumption of tokens in the AWARD model is determined by autonomic activities that may have different paces for executing successive iterations.

Due to the above issues, as depicted in Figure 4.3, it becomes necessary to ensure a global coordination between the activities involved $(A_1,...,A_j)$, in order to guarantee that at any global observation point (K in the Figure 4.3) the reconfiguration plan for a transition Rcf_K can be applied in a such a way that the new activity contexts $(Ctx(A_1)_{after},...,Ctx(A_j)_{after})$ correspond to a new workflow configuration Wcf_{K+1}, which preserves the semantics of the AWARD model (according to above Definitions 4.4 and 4.5).

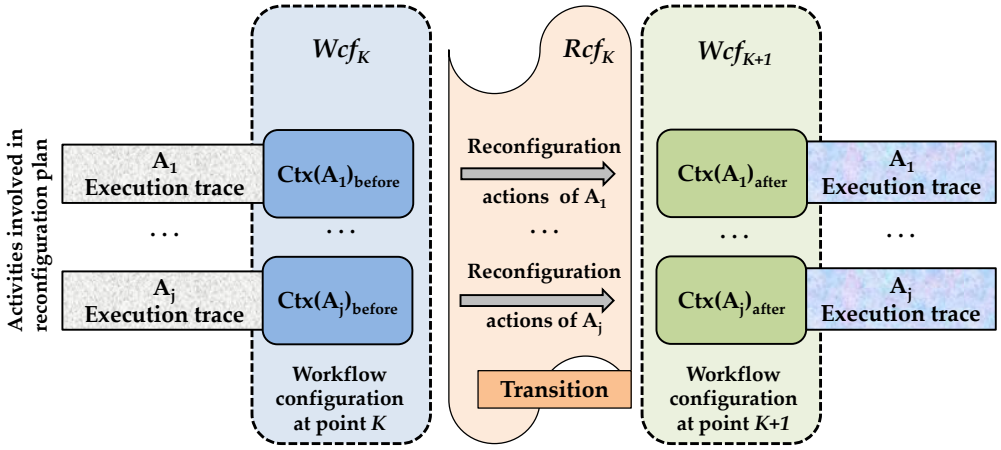


Figure 4.3: Reconfiguration plan transition at observation point K

The global coordination between the activities involved in the reconfiguration plan should ensure the following conditions:

1. *Atomicity:* The transition between workflow configurations should be atomic (all or nothing);

2. *Serialization:* In the presence of multiple requests for concurrent reconfiguration plans affecting the same activities, their global effect must be equivalent to a sequential execution of the reconfiguration plans;

3. *Consistency:* The Rcf_K reconfiguration plan should be consistent according to Definition 4.5, that is, both Wcf_K and Wcf_{K+1} configurations should satisfy the AWARD soundness properties of Definition 4.4.

Note that, due to the asynchronous and autonomic characteristics of the AWARD model, as depicted in Figure 4.4, a long-running workflow with a sound current configuration Wcf can be subject to distinct reconfiguration plans $(Rcf_1,...,Rcf_n)$ that could lead

to a set of distinct new workflow configurations ($Wcf_1, ..., Wcf_n$), where some of them are sound and others are unsound, depending on the fulfillment of the above conditions, namely ensuring the soundness properties of Definition 4.4.

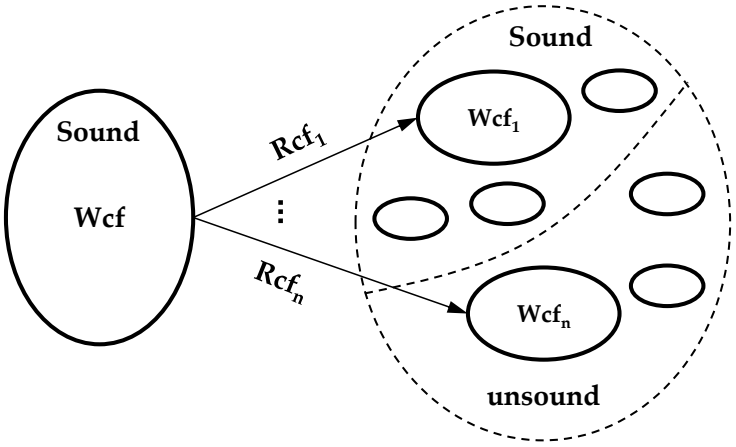


Figure 4.4: Applying distinct reconfiguration plans to a sound workflow configuration

Due to the AWARD characteristics for supporting autonomic activities that are running asynchronously without a centralized control, there is a concern related to the verification of the soundness properties (as in Definition 4.4) that should be preserved when applying a reconfiguration plan.

On one hand, the soundness verification should be delegated to an external entity that acts as a global observer regarding the execution context of the entire workflow, in order to validate if the transition $Wcf_i \xrightarrow{Rcf_i} Wcf_{i+1}$ corresponds to a consistent reconfiguration plan (as in Definition 4.5). By relying on an external entity for the verification of the consistency of reconfiguration plans, this approach provides great flexibility as it allows the above verification to be performed according to the application scenario.

On the other hand, from the point of view of the AWARD model the above decision is compatible with the approach of providing a flexible model for dynamic reconfiguration, which relies on a basic set of AWARD dynamic reconfiguration operators.

However, some basic mechanisms must be supported by the AWARD machine in order to ensure that the *Atomicity* and *Serialization* conditions are enforced, and also that each workflow activity in a long-running workflow executes the corresponding reconfiguration operators at an adequate iteration. The latter guarantee must be based on a common/global iteration agreement, collectively reached by all the activities involved in the scope of a reconfiguration plan.

In the following, we discuss the issue of the verification of reconfiguration plans based on an external entity, although this is left out of the scope of this dissertation. Then we discuss the basic mechanisms supported by the AWARD machine concerning the *Atomicity* and the common/global iteration agreement.

✧ **Verifying the consistency of a reconfiguration plan by an external entity**

The approach that relies on an external entity for verifying the consistency of a reconfiguration plan is schematically illustrated in Figure 4.5.

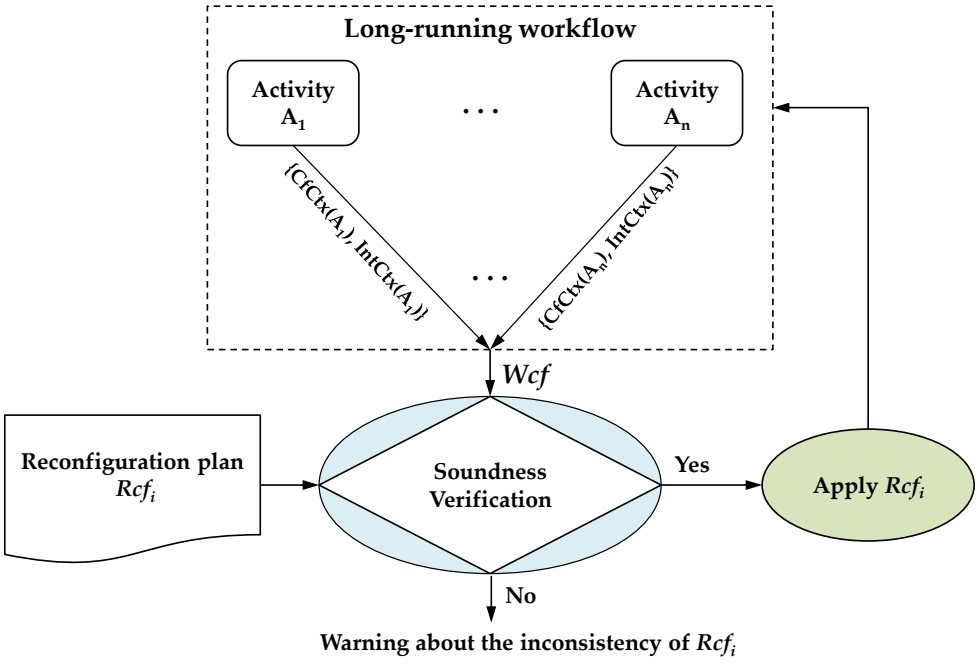


Figure 4.5: Soundness verification before applying a reconfiguration plan

During the execution of a long-running workflow and before applying a reconfiguration plan (Rcf_i), an external entity analyses the workflow configuration (Wcf) defined by the contexts of all workflow activities and verifies if the resulting workflow configuration is compliant to the soundness properties.

If the external entity, which performs the soundness verification, concludes for the compliance, it then enables the application of the reconfiguration plan (Rcf_i), otherwise it notifies the user about the resulting unsound configuration.

The AWARD machine is neutral in relation to the strategies used by external entities for prevalidating the compliance to soundness properties. This neutrality introduces great flexibility in the AWARD model since it allows workflow developers to define the more adequate strategies to enforce soundness properties according to their knowledge about the application domain. For example, if a reconfiguration plan replaces the activity $Tasks$ with new algorithms for changing the workflow functional requirements, the resulting processed data for the new configuration can be different for the same input data. It is up to the application developer to verify if this is acceptable and correct. However, the new workflow configuration would also be sound by Definition 4.4.

✧ **Consistency of the common/global iteration agreement**

The AWARD model only provides the basic mechanism for the activities to reach an agreement to choose the earliest global iteration between all activities involved such that

a reconfiguration plan is applied. The activities are only responsible for providing this basic guarantee by participating in a iteration agreement based on a two phase commit protocol and by ensuring that their reconfiguration actions are atomically processed. The point for atomically processing the dynamic reconfiguration operators is at the beginning of the iteration whose number (K) resulted from the agreement between the activities involved in the reconfiguration plan.

Definition 4.6: ***Global iteration agreement***

The execution of a reconfiguration plan Rcf_i satisfies the common/global iteration agreement if it is applied to a sound workflow configuration Wcf_i at an iteration number K, resulting from an agreement of all activities $(A_1,...,A_n)$ involved in the Rcf_i scope, in order to choose K as the earliest global iteration number, given by $K = maximum(\{It_1,...,It_n\})$ where It_i is the proposed iteration number by activity A_i, with $1 \leqslant i \leqslant n$. If all $It_i > 0$ the execution of the reconfiguration plan succeeds, otherwise if any $It_i = -1$ (the activity A_i can not apply its reconfiguration actions), then $K = -1$ and in this case the execution of the reconfiguration plan fails, with no effects.

Definition 4.7: ***Iteration proposed by activities***

If an activity A_i, $(1 \leqslant i \leqslant n)$ is in a fault state $(faultIn, faultTask, faultOut)$ the proposed iteration number It_i is equal to its current iteration number; otherwise It_i is equal to the current iteration number plus 1. If the activity can not apply its reconfiguration, then It_i is equal to -1.

✧ **The earliest global iteration (K)**

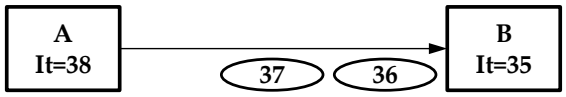
The workflow activities can be running asynchronously with different current iteration numbers. According to the AWARD model the current iteration number is used to mark the tokens allowing the coordinated token processing on destination input ports. Therefore for maintaining the consistency of links (properties 2 and 3 of Definition 4.4 on page 94) between activities involved in a reconfiguration plan it is necessary to apply the reconfiguration plan at a common iteration number agreed between all activities involved. This needs to be inferred by the basic mechanisms of the AWARD machine for supporting dynamic reconfigurations.

From the point of view of each individual activity the earliest point for performing reconfigurations is before the activity starts the next iteration. Thus for applying reconfiguration plans the adequate iteration number is the current iteration number plus 1. However, if an activity enters a fault state $(faultIn, faultTask, faultOut)$ then for trying fault recovery of this activity the reconfiguration plan must be applied at the current iteration number. Therefore before applying a reconfiguration plan each activity must supply its adequate iteration number for performing the actions involved in the reconfiguration plan.

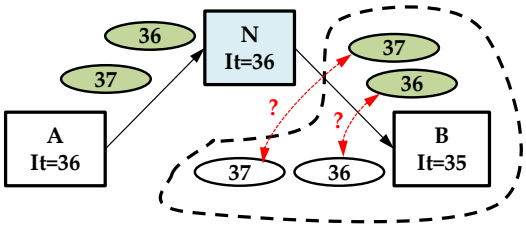
Considering $IterSet = \{it_1, ..., it_n\}$ as the set of adequate iteration numbers obtained from all activities involved in the reconfiguration plan, a simple way to choose the best common iteration number, denoted by $BestIt$, is choosing the earliest global workflow iteration calculated as the maximum iteration number of the set $\{it_1, ..., it_n\}$, that is, $BestIt = maximum(\{it_1, ..., it_n\})$. Otherwise, if the chosen iteration was less than $BestIt$, then before applying the reconfiguration plan for some activities it would be necessary to perform complex rollback actions, for instance based on execution logs in order to repeat the processing already done in previous iterations.

In order to illustrate the need to choose $BestIt$ as the best common iteration number, $BestIt = maximum(\{it_1, ..., it_n\})$, let us consider the workflow reconfiguration example illustrated in Figure 4.6.

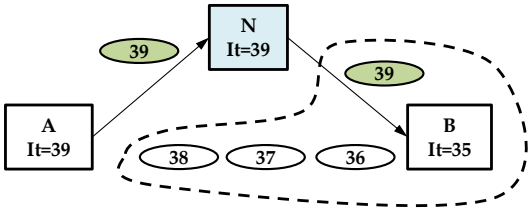
The long-running workflow has two activities A and B where the A activity produces tokens on its output port, which are received by the B activity on its input port configured in the $Iteration$ input mode. Assuming that the two activities have different processing speeds, Figure 4.6(a) depicts an execution point where the A activity is in the 38^{th} iteration but the B activity is still in the 35^{th} iteration so the link from A to B has two pending tokens for delivery, which were already produced by the A activity, respectively, in its iteration numbers 36 and 37.



(a) Long-running workflow with activities A and B in distinct iterations (38, 35)



(b) Applying a reconfiguration at iteration 36



(c) Applying the reconfiguration at iteration 39 as a result of $maximum(38+1, 0+1)$

Figure 4.6: A reconfiguration plan which introduces the new activity N

Let us assume a reconfiguration plan to introduce the new activity N between A and B. Such scenario does not involve the B activity because it is not affected. In fact, due to the autonomic characteristics of an AWARD activity the source of tokens sent to input ports is not known.

However, let us arbitrarily assume two possibilities to make a choice of an iteration number to apply the reconfiguration plan.

One possibility is to apply the reconfiguration plan at iteration number 36, as depicted in Figure 4.6(b), where like the A activity the new activity N would produce two tokens

for its iteration numbers 36 and 37.

Other possibility, as depicted in Figure 4.6(c), is to apply the reconfiguration at iteration number 39 following the rule of choosing $BestIt = maximum(\{38+1, 0+1\}) = 39$, that is, the involved A and N activities proposed respectively (38+1) and (0+1) as their current iteration numbers plus 1.

Comparing the two possibilities we can understand the problem of choosing an iteration number less than $BestIt$.

In Figure 4.6(b), the new activity N would produce two tokens for its iteration numbers 36 and 37, which leads to an ambiguity regarding the tokens processing because, before applying the reconfiguration plan, the activity A had already produced tokens at its iteration numbers 36 and 37 to be consumed by the input port of activity B. Besides, in this case, the A activity would have to roll back to the 36^{th} iteration.

In Figure 4.6(c) there is no such ambiguity because the activity B consumes its tokens independently of the source of each token.

Therefore, before applying a reconfiguration plan it is necessary to reach an agreement between the affected workflow activities in order to find the $BestIt$ iteration number as the earliest global workflow iteration number where all activities must apply the actions of the reconfiguration plan. If the agreement succeeds by finding a $BestIt$ iteration number, the reconfiguration plan is committed; otherwise if the agreement is not possible, the reconfiguration plan is cancelled. For instance, if one affected activity has already terminated because the last iteration was reached the agreement is not possible.

In the following (Section 4.2), we present the operators that can be used by workflow developers to specify dynamic reconfiguration plans.

In Section 4.3, we describe the extensions to the *Autonomic Controller* of the AWARD machine to support the execution of the dynamic reconfiguration operators.

4.2 Dynamic Reconfigurations: Programmer's View

Workflows with large number of iterations can run for a long time during which the workflow programmers may decide to change something in the structure or in the behavior of the workflows. This decision can arise from the analysis of intermediate results or even because some requirements of the workflows have changed.

After the decision for changing a workflow a programmer needs preparing a reconfiguration plan as a sequence of actions to be submitted to the workflow activities involved in the reconfiguration plan.

The programmer's view for specifying a reconfiguration plan consists of the preparation of a script where the programmer specifies operators at the following levels:

1. **Compound:** Operators for which the AWARD machine ensures atomicity of the entire reconfiguration plan and the common iteration agreement where all involved activities only apply their specific individual reconfiguration actions at an iteration number K previously agreed upon (according to Definition 4.6 on page 98);

2. **Elementary:** Operators to modify the *AWA Context* of the activities, including a particular operator to launch a new workflow activity.

As depicted in Figure 4.7 the interpretation of the script that specifies a reconfiguration plan is executed with the semantics of a two-phase commit protocol. After the involved activities agree to apply their specific actions at an iteration K, the set of distinct changes in multiple workflow activities is applied as an atomic operation with an *"all or nothing"* effect.

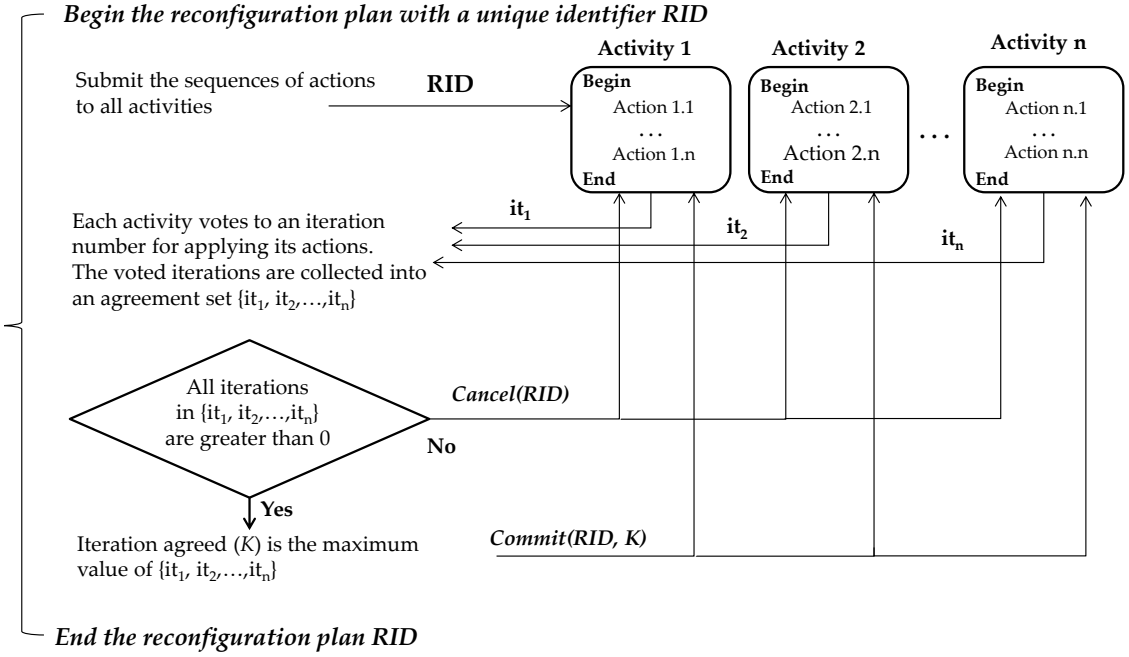


Figure 4.7: Execution of a reconfiguration plan involving multiple activities

In a first phase, the script interpreter acts as the coordinator of the commit protocol and submits sequences of actions as blocks of operators to all activities involved. Each block is delimited by *Begin/End* keywords, marked with a unique global identifier (RID) of the reconfiguration plan requested, and is stored internally in each activity. In a second phase the script interpreter collects an agreement set $AgSet = \{It_1, ..., It_n\}$ containing the iteration numbers voted by the activities for applying their specific reconfiguration sequences (as in Definition 4.6 on page 98). If some activities propose invalid iteration numbers ($It_i = -1$), for instance, if an activity involved is in its last iteration, the reconfiguration plan is cancelled by sending the command *Cancel(RID)* to all activities. Otherwise the script interpreter proceeds with the reconfiguration plan by sending the command *Commit(RID, K)* to all activities, indicating that each activity must perform its sequence of actions at the K agreed iteration number, which is the maximum iteration number of the agreement set ($AgSet$), that is, $K = maximum(\{It_1, ..., It_n\})$.

In this way, the script interpreter ensures that a reconfiguration plan is atomic and synchronized by an iteration number K agreed by all activities involved.

Each activity autonomously and atomically applies its sequence of actions when it reaches the iteration number K previously agreed.

When an activity reaches an iteration, with $It_i > 0$, that it has proposed as a vote associated with an RID reconfiguration plan, it waits until receiving a *Cancel* or $Commit(K)$ command related to the RID reconfiguration plan. Otherwise if the activity has proposed ($It_i = -1$) the activity does not wait and proceeds with its execution.

After having received a *Cancel* command the activity discards the reconfiguration plan locally and proceeds. If it received a $Commit(K)$ command the activity proceeds the execution unchanged until it reaches the iteration number (K) that was agreed upon, and then it applies the corresponding local reconfiguration actions.

In order to specify the scripts for performing reconfiguration plans, the workflow developers rely on a set of dynamic reconfiguration operators. These operators are provided by the *Dynamic API* interface, and encapsulate the interactions between a script interpreter and the dynamic reconfiguration handlers in the autonomic controllers of the activities involved in the reconfiguration plan (as depicted in Figure 3.12 on page 75 of Chapter 3).

4.2.1 Operators for Dynamic Reconfiguration

The AWARD model for dynamic reconfigurations allows performing reconfiguration plans involving one or multiple activities using scripts. The reconfiguration script is a sequence of operators which are processed by a script interpreter that triggers the interactions with all activities involved in the reconfiguration plan (Figure 4.7). The specification of a script is enclosed by $BeginReConfiguration$ and $EndReConfiguration$ compound operators that enforce the atomicity of the reconfiguration plan.

Listing 4.1: A reconfiguration script involving multiple activities

```
1  RID=BeginReConfiguration(Awa1,...,AwaN)
2
3  BeginAwaReConfig(RID, Awa1) // begin of Awa1 reconfiguration block
4    // Sequence of operators to reconfigure the activity Awa1
5    . . .
6  It1=EndAwaReConfig(RID, Awa1) // end of Awa1 reconfiguration block
7  . . .
8  BeginAwaReConfig(RID, AwaN) // begin of AwaN reconfiguration block
9    // Sequence of operators to reconfigure the activity AwaN
10   . . .
11 Itn=EndAwaReConfig(RID, AwaN) // end of AwaN reconfiguration block
12 AgreementSet={It1,...,Itn} // The set of proposed iteration numbers
13 K=EndReConfiguration(RID, AgreementSet)  // returns the iteration agreed upon
```

As shown in Listing 4.1 a script is built as a sequence of reconfiguration blocks for the set of activities involved $\{Awa1,\ldots,AwaN\}$.

The atomic reconfiguration block as a sequence of operators for performing structural and behavioral reconfigurations of each individual activity is enclosed between $BeginAwaReConfig$ and $EndAwaReConfig$ compound operators.

In the following we describe the set of dynamic reconfiguration operators supported by the AWARD model. In Section 4.3.3 we formally describe each operator as performing a transition from an old activity *AWA Context* to a new activity *AWA Context*, which reflects the changes performed by the operator.

The operators description is organized in two groups:

1. *Compound operators:* A set of operators for defining the scope of a reconfiguration plan, and the activity reconfiguration blocks;

2. *Elementary operators:* A set of operators for performing structural and behavioral workflow changes and to manage the life-cycle of an activity, including an operator to launch a new workflow activity.

➦ Compound Operators

As illustrated in the script of Listing 4.1, the compound operators are *Begin/End* pairs for specifying a reconfiguration plan involving multiple workflow activities and a reconfiguration block applied to each activity. The following compound operators are supported:

- **BeginReConfiguration:** This operator starts the script execution by generating a unique identifier (RID) of this reconfiguration plan. This RID identifier is included as an input argument to each activity reconfiguration block that is sent by the script interpreter to each individual activity;

- **BeginAwaReConfig:** This operator starts the reconfiguration block of each individual activity involved. The script interpreter signals the AWA activity to be prepared

to receive a sequence of dynamic reconfiguration operators. The activity updates its internal data structures for storing the sequence of operators;

- **EndAwaReConfig:** This operator ends the reconfiguration block of each individual activity involved. The activity returns to the script interpreter a proposed iteration number (It_i) voted by the activity for applying its reconfiguration block. The script interpreter collects all proposed iterations numbers in the agreement set $AgSet = \{It_1, ..., It_n\}$ to be used for calculating the iteration number as the best iteration for applying the reconfiguration plan;

- **EndReConfiguration:** This operator marks the end of the script execution and enforces the atomicity of the reconfiguration plan. If some activity has proposed an iteration number ($It_i = -1$), indicating its impossibility to apply the reconfiguration actions, the reconfiguration plan is cancelled and all activities discard their corresponding reconfiguration blocks. Otherwise, after all activities have replied with proposed iteration numbers $It_i > 0$, this operator calculates the earliest global iteration number as $K = maximum(\{It_1, ..., It_n\})$ and sends commands to all activities for committing the RID reconfiguration plan to be applied at iteration K. If the reconfiguration plan commit succeeds, this operator returns the agreed iteration number (K) to the invoking script interpreter, otherwise it returns -1 for indicating the cancelation of the reconfiguration plan.

➦ **Elementary operators**

Elementary operators are used enclosed by an activity reconfiguration bock, except for the $LaunchActivity$ operator, which is used to launch a new workflow activity.

- **LaunchActivity:** This operator launches the execution of a new activity to change the structure of the workflow. The operator allows launching the new activity on local or remote computing nodes. On being launched the new activity waits for a reconfiguration block *BeginAwaReConfig/EndAwaReConfig* in order to participate in the reconfiguration plan agreement. Accordingly, after the $LaunchActivity$ operator the script must include a reconfiguration block in order to send a starting command ($StartExec$ operator) for the new activity, as shown in the example of Listing 4.2 on page 106. As we shall see in Section 4.3.2 the operator $LaunchActivity$ introduces the need to extend the *State Machine* of the *Autonomic Controller* with a new state for ensuring the reconfiguration of an activity, which was launched during a reconfiguration plan.

✧ **Structural and behavioral operators**

An activity reconfiguration block specifies a sequence of operators to perform dynamic reconfigurations related to structural and behavioral changes to be applied to each individual activity. The operators supported by the AWARD model are:

- **ChangeParameters:** This operator changes the activity *Parameters*. This is a basic but useful operator for changing the behavior of the activity *Task*, for instance, for supporting computational steering;
- **ChangeTask:** This operator changes the *Task* of an activity allowing it to change its behavior by executing a new algorithm, for instance to recover from failures or to introduce algorithm optimizations;
- **CreateInput:** This operator creates a new activity input port according to the new workflow structure resulting from the reconfiguration plan;
- **ChangeInputOrder:** This operator changes the order mode for token consumption of an input port following three possible order modes (as in Definition 3.7 on page 60): *Iteration* - the order is based on the iteration number; *Sequence* - the order is based on a sequence number generated by predecessor activities; or *Any* - tokens are consumed non deterministically in any order;
- **CreateOutput:** This operator creates a new activity output port according to the new workflow structure resulting from the reconfiguration plan;
- **ChangeOutputLink:** This operator changes the complete destination list of an output port by changing the links associated to the output port;
- **AddOutputLink:** This operator adds one more link to the destination list of an output port by adding a new destination input port connected to the output port;
- **ChangeOutputStrategy:** This operator changes the output port strategy for sending tokens following three possible modes (as Definition 3.8 on page 60): *Single* - when the output port is connected to a simple link and on each iteration a single token is sent to the single destination input port; *Replicate* - the output port is connected to a multi-link and each token is replicated and sent to all connected destination input ports; or *RoundRobin* - the output port is connected to a multi-link and the tokens marked with a sequence number are alternatively sent to each of the connected destination input ports;
- **ChangeInputState:** This operator changes the input port state (as in Definition 3.15 on page 64): *Enable*, *Disable*, *EnableFeedback*;
- **ChangeOutputState:** This operator changes the output port state (as in Definition 3.15 on page 64): *Enable*, *Disable*, *EnableFeedback*;
- **ChangeMaxIterations:** This operator changes the maximum number of iterations of an activity;
- **ChangeMappingInputs:** This operator changes the way how data from activity input ports are mapped to *Task Arguments*;

- **ChangeMappingOutputs:** An activity *Task* can return a sequence of results. This operator changes the way how *Task Results* are mapped to the activity output ports.

✧ **Activity Life-cycle Operators**

The following set of operators can be used to control the life-cycle of an activity:

- **StartExec:** This operator starts the execution of an activity, which was launched using a *LaunchActivity* operator. The operator ensures that the new activity starts its execution at the iteration resulting from the agreement achieved by the reconfiguration plan as illustrated by activity (N) in example of Figure 4.6 on page 99;
- **Suspend/Resume:** This operator suspends or resumes the execution of an activity;
- **Terminate:** This operator explicitly and synchronously terminates the activity execution. Depending on the application scenario and the workflow structure the activities are autonomous running at their own paces so it may be impossible to obtain an agreement for the iteration number where all activities must finish together. Therefore for terminating activities individual activity reconfiguration blocks should be submitted including one *Terminate* operator, as illustrated in Listing 4.3;
- **RetryAfterFaultIn:** This operator allows an activity to get back to the *input* state for retrying the execution at the same point where the failure occurred to proceed assuming that recovery actions related to the AWARD Space were performed;
- **RetryAfterFaultTask:** This operator allows an activity to get back to the *invoke* state for retrying the *Task* invocation assuming that recovery actions were performed, for instance a new implementation of the *Task* was specified using the *ChangeTask* operator to correct the failure;
- **RetryAfterFaultOut:** This operator allows an activity to get back to the *output* state for retrying to proceed the execution at the same point where the failure occurred assuming that recovery actions related to the AWARD Space were performed;
- **GetAwaContext:** This operator returns the *AWA Context* ($Ctx(A)$) of an activity. This operator can be used by any tool developed for dynamically monitoring the internal state of an AWA activity during its execution.

Listing 4.2: A reconfiguration block after launching a new activity

```
1 BeginAwaReConfig(RID, NewAwa)
2    StartExec(RID,NewAwa) // Start the execution of the NewAwa activity
3 IterProposed=EndAwaReConfig(RID, NewAwa)
```

Listing 4.3: Reconfiguration script to explicit synchronous termination of an activity

```
1 BeginAwaReConfig(RID, AwaName)
2    Terminate(RID,AwaName) //Terminate the execution of the AwaName activity
3 IterProposed=EndAwaReConfig(RID, AwaName)
```

The operators *LaunchActivity*, *CreateInput* and *CreateOutput* require the enforcement of the uniqueness of the names assigned to the new activity and the new input and output ports.

A complete list of the above operators and their detailed parameters is presented in Figure 5.13 on page 167.

4.2.2 An Example of a Workflow Dynamic Reconfiguration

To illustrate a complete example of a dynamic reconfiguration let us consider the workflow in Figure 4.8 (based on the example of Figure 3.8 on page 64 without the loop with a feedback activity). Assuming that this workflow has an infinite number of iterations, now we consider that when analyzing intermediate results during the workflow execution, the workflow developer decides to change the workflow by introducing the feedback loop as presented in Figure 4.9.

Recalling what has already been described in Section 3.3.3 (Specifying a concrete workflow, on page 65) the *Feedback* activity analyses the array of features as strings received on I_1F input port and produces, on its O_1F output port, a token with a data filter criterion stored as a data type with the absolute name *wkf.FilterCriteria*. It is also important to recall that the I_1A input port needs to be set initially at the *EnableFeedback* state, passing automatically to the *Enable* state at the next iteration. However, unlike the example of Figure 3.8, the *Feedback* activity will be now dynamically launched during the workflow reconfiguration.

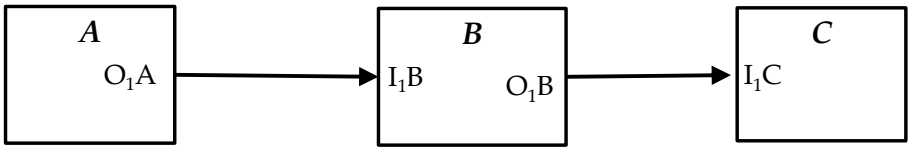


Figure 4.8: A workflow example without loops

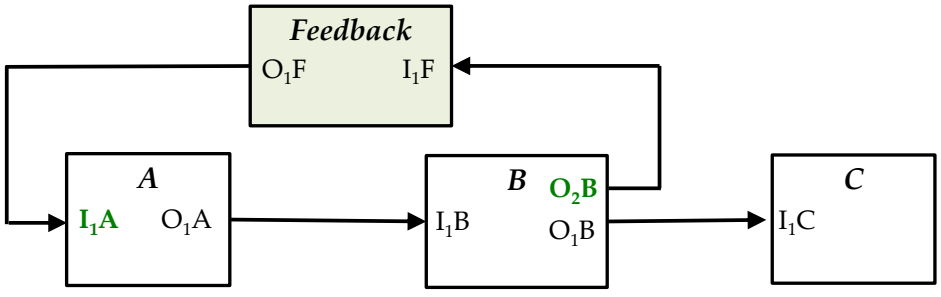


Figure 4.9: Change the workflow of Figure 4.8 with a feedback loop

In order to achieve the above dynamic reconfiguration the workflow developer needs to prepare a script with the reconfiguration plan containing the following sequence of actions:

Listing 4.4: The script to change the workflow of Figure 4.8 to the workflow of Figure 4.9

```
int RID = BeginReConfiguration("A", "B", "Feedback")
 BeginAwaReConfig(RID, "A")
   CreateInput(RID, "A", "I1A", "EnableFeedback", "wkf.FilterCriteria")
   ChangeMappingInputs(RID, "A", new String[]{"I1A"});
   ChangeTask(RID, "A", "wkf.TaskAwith1Arg");
 int it1 = EndAwaReConfig(RID, "A")
 BeginAwaReConfig(RID, "B")
   CreateOutput(RID,"B","02B","Enable","String[]","Single",new String[]{"I1F"})
   ChangeTask(RID, "B", "wkf.TaskBwith2Results")
   ChangeMappingOutputs(RID, "B", new String[]{"01B", "02B"})
 int it2 = EndAwaReConfig(RID, "B")
 LaunchActivity("Feedback","FeedbackSpec.xml")
 BeginAwaReConfig(RID, "Feedback")
   StartExec(RID, "Feedback")
 int it3 = EndAwaReConfig(RID, "Feedback")
 AgreementSet={it1,it2,it3}
int K=EndReConfiguration(RID, AgreementSet)
```

1. Create an input port in the A activity, initially in state $EnableFeedback$, and a token type $wkf.FilterCriteria$;

2. Since the A activity has now an input port it is necessary to change the inputs mapping function of the A activity to define the new map pair $(0, I_1A)$. This pair associates the $Task$ argument in index 0 of the A activity to its input port named I_1A;

3. Since the $Task$ of the A activity has now an argument in index 0 to be processed it is necessary to change the $Task$ of the A activity to process one more argument;

4. Create a new output port, named O_2B in the B activity initially in state $Enable$ with a token type as an array of features as strings and a single link to the I_1F destination port;

5. Change the $Task$ of the B activity to produce two results to be mapped to the two output ports O_1B and O_2B;

6. Change the outputs mapping function of the B activity to map both $Task\ Results$ to the two outputs O_1B and O_2B;

7. Launch the new $Feedback$ activity (specified in the file $FeedbackSpec.xml$) to run in some available computing node.

The above described script of the reconfiguration plan is presented in Listing 4.4 demonstrating in detail how dynamic operators are used to perform dynamic reconfigurations.

The script starts with the $BeginReConfiguration$ operator, which gets a global unique identifier (RID) for marking all operators related to the reconfiguration plan. The script

shows that each activity involved (A, B and $Feedback$) receives a sequence of operators to perform the actions needed in each activity reconfiguration. The sequence of operators of each activity is an atomic block enclosed between the $BeginAwaReConfig$ operator and the $EndAwaReConfig$ operator. The latter operator $EndAwaReConfig$ returns the iteration number that each activity proposes as the earliest iteration number where it can atomically apply its reconfiguration block. The agreement set of iterations proposed by the three activities involved is denoted by $AgreementSet$ and is equal to $\{it1, it2, it3\}$, which includes the iterations proposed respectively by A, B and $Feedback$ activities.

The final operator $EndReConfiguration$ performs the following steps: i) If the agreement set ($AgreementSet$) contains some invalid iteration number (-1), the earliest global iteration K is invalid (-1), otherwise K is calculated as $maximum(it1, it2, it3)$; ii) If K is invalid the reconfiguration plan RID is cancelled, otherwise a coordinated commit is issued involving the A, B and $Feedback$ activities, which must apply the reconfiguration block marked with RID at the iteration defined by K; and iii) The operator returns the value of K which can be used to report the successful or unsuccessful result of the reconfiguration plan.

The $LaunchActivity$ operator receives, as a second argument, the name of the file $FeedbackSpec.xml$, containing the specification of the new $Feedback$ activity as presented in Table 4.1 and invokes the AWARD tool ($AwardLaunchAWA.jar$ presented in Chapter 5, Section 5.9) used to launch an AWA activity in the available computing nodes.

Table 4.1: Specification of the *Feedback* activity

AWA	*Name*	**Feedback**				
	InitialState	**WaitConfig**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1F	**Enable**	**java.lang.String[]**	**Iteration**	
	Outputs	*Name*	*State*	*TokenType*	*OMode*	*SendTo*
		O1F	**EnableFeedback**	**wkf.FilterCriteria**	**Single**	**I1A**
	Task	*Parameter*		**"server=S;database=DBHist;"**		
		SoftwareComponent		**wkf.TaskDataFiltering**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1F	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				0	**O1F**	

The specification of the new $Feedback$ activity is basically the same as the specification of the $Feedback$ activity in the workflow presented in Table 3.1 of Section 3.3.3 (Specifying a concrete workflow, on page 65) except for the new $InitialState$ field, which contains the value $WaitConfig$. In Section 4.3.2 the need for this new field is explained when discussing how the *State Machine* of the *Autonomic Controller* controls the processing of dynamic reconfiguration operators.

4.3 The AWARD Machine and Dynamic Reconfigurations

After having presented how programmers can prepare scripts with reconfiguration plans for dynamically changing a long-running workflow, this section describes the operational view of the AWARD machine for supporting dynamic reconfigurations.

The interactions with the AWARD machine related to supporting dynamic reconfiguration are based on the publishing/subscribing model. As presented in Section 3.4 in addition to supporting the links for connecting the input and output ports, the AWARD Space has properties for allowing dynamic interactions based on the publishing/subscribing model.

Therefore for each AWA activity the *Autonomic Controller* (AC) of the AWARD machine registers a *Dynamic Reconfiguration Handler* (DRH) on the AWARD Space for subscribing to the asynchronous events related to dynamic reconfiguration operators.

The AWARD model is neutral regarding the way how these events are published into the AWARD Space, which allows flexible implementation approaches. In terms of the operational view this approach relies on a software library addressing two goals:

1. Providing a reusable software library with an application programming interface (named by *Dynamic API*) for exposing the set of reconfiguration operators;

2. Publishing, into the AWARD Space, the events corresponding to the requests for applying the distinct dynamic reconfiguration operators.

In the following (Section 4.3.1), we describe how reconfiguration requests to an AWA activity are handled by interactions between the *Dynamic API*, the AWARD Space and the *Dynamic Reconfiguration Handler* (DRH) within the *Autonomic Controller* (AC). In Section 4.3.2 we present the extensions to the *State Machine* of the *Autonomic Controller* (AC) for supporting the new states where dynamic reconfiguration operators are processed. Finally in Section 4.3.3 the semantics of each dynamic operator is described in terms of context transitions expressing how the *AWA Context* is changed.

4.3.1 Handling Reconfiguration Requests through the Dynamic API

Figure 4.10 illustrates the operational view of the interactions between the *Dynamic API*, the AWARD Space and the *Dynamic Reconfiguration Handler* (DRH) within the *Autonomic Controller* (AC). The *Dynamic API* provides an application programming interface (API) exposing the reconfiguration operators for allowing the submission of reconfiguration plans. During the initialization of each *Autonomic Controller*, a *Dynamic Reconfiguration Handler* (DRH) is registered into the AWARD Space in order to subscribe to asynchronous events. Each event is represented by a tuple $(RID, Event, AWAname, argsList)$, where RID is a reconfiguration plan identifier, $Event$ identifies the event, $AWAname$ identifies the activity involved in the event, and $argsList$ is a list of $Arguments$ according to the event.

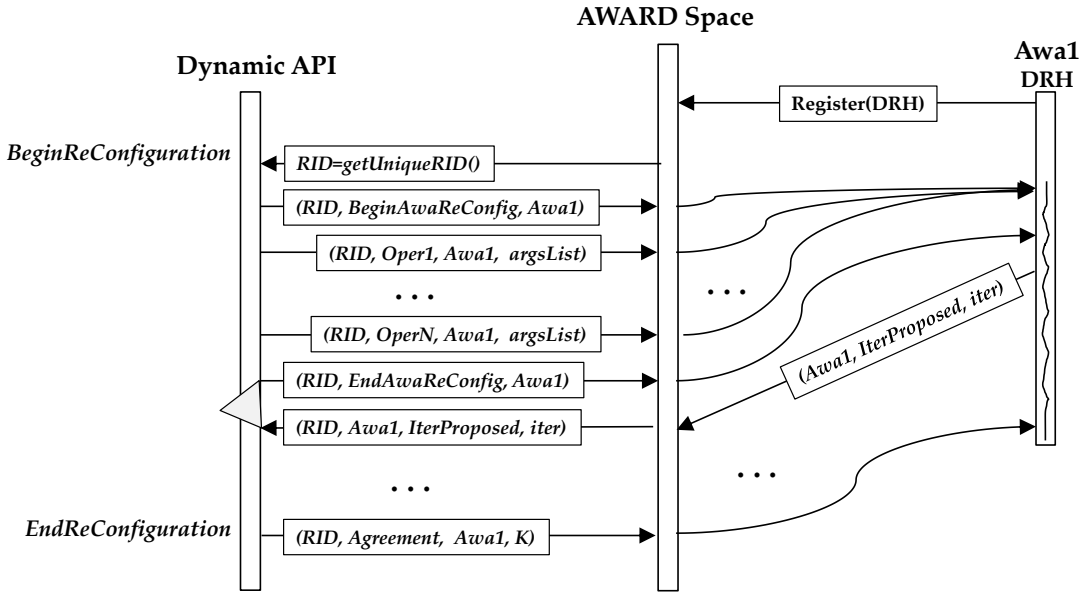


Figure 4.10: The operational interactions for applying dynamic reconfigurations

A reconfiguration plan starts by invoking the $BeginReConfiguration$ operator through the *Dynamic API* to get a unique identifier (RID) of the reconfiguration plan, as presented in Section 4.2 (Figure 4.7 on page 101).

For each operator the *Dynamic API* allows to publish events into the AWARD Space as tuples with the $(RID, Event, AWAname, argsList)$ pattern subscribed by the *Dynamic Reconfiguration Handler* (DRH). Then the DRH handles and stores the sequence of operators until it receives the $EndAwaReConfig$ operator, which implies to return an iteration number as a proposal to perform the AWA activity reconfiguration. The DRH handler gets the activity current iteration number from the *AWA Context* and proposes this iteration number according to Definition 4.7 on page 98, thus indicating that from the point of view of this AWA activity ($Awa1$), the dynamic reconfiguration can take place as soon as possible, that is normally at the next iteration or at the current iteration in the case of the activity being in a fault state.

The iteration proposal, including the iteration number to propose ($iter$), is published into the AWARD Space as a tuple $(RID, AWAname, IterProposed, iter)$ that will be consumed during the processing of the $EndAwaReConfig$ operator.

When the $EndReConfiguration$ operator is processed the earliest global iteration agreed, denoted by K, is calculated as the maximum of all iteration numbers proposed by all activities involved, assuming these numbers are all valid iteration numbers greater than zero. Otherwise, if any activity has proposed an iteration number invalid (-1) to signal that it can not participate in the reconfiguration plan, then $K = -1$. Afterwards K is published into the AWARD Space as a tuple $(RID, Agreement, AWAname, K)$. Each DRH handler, in each AWA activity consumes this tuple and if K is greater than zero it marks the corresponding AWA activity reconfiguration sequence as ready to be processed

when its *State Machine* reaches the iteration agreed; otherwise if K is equal to -1, the reconfiguration was not committed and the reconfiguration sequence marked with RID is discarded by each *Dynamic Reconfiguration Handler* of the activities involved in the reconfiguration plan.

As an example let us consider that an AWA activity reconfiguration sequence is handled by its DRH handler during iteration number 14. As illustrated in Figure 4.11 when the handler receives the $EndAwaReConfig$ operator, it proposes the iteration number 15 as a possible iteration number to apply its reconfiguration. Later when the DRH handler receives the tuple with the agreement to the reconfiguration identified by RID it marks the corresponding AWA activity reconfiguration sequence to take place at iteration 25, which was the earliest global iteration K agreed by all activities involved. Later, when the *State Machine* reaches the 25^{th}iteration, the reconfiguration actions will be applied.

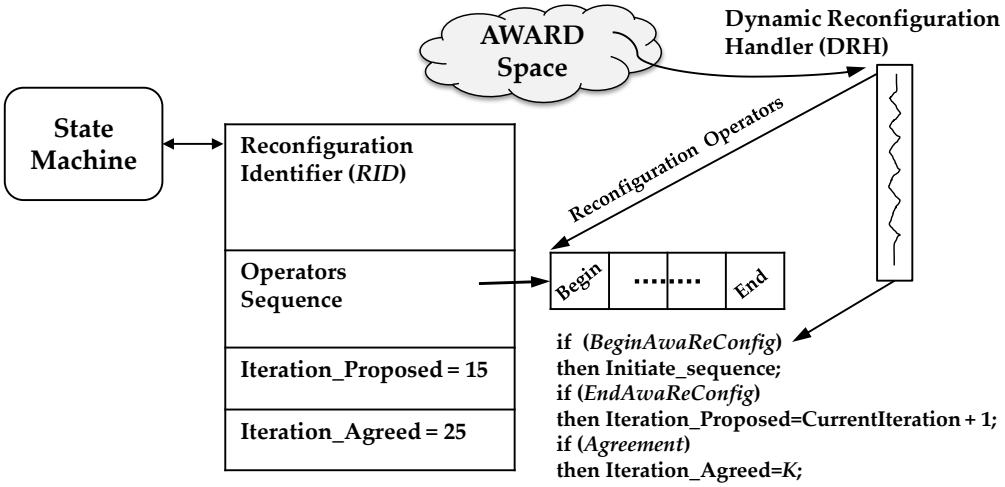


Figure 4.11: Reconfiguration sequence to be processed by an AWA activity

In order to support dynamic reconfigurations, the AWARD machine needs extensions to the *State Machine* of the *Autonomic Controller* (AC) presented in Section 3.4.5.3. These extensions must meet the following requirements:

1. When the *State Machine* reaches the $idle$ state at a previously proposed iteration number, it needs to wait for the confirmation of the agreement completion before proceeding;

2. When the *State Machine* reaches the $idle$ state at an iteration number already agreed upon to apply a given reconfiguration, it needs to enter a new state ($Config$) in order to process the corresponding sequence of reconfiguration operations.

Due to the RID uniqueness of each reconfiguration plan, an AWA activity can agree to submit multiple reconfiguration sequences at the same iteration number. In this case, each reconfiguration sequence is applied according to the order of the reconfiguration identifier (RID).

4.3.2 Extended State Machine for Supporting Dynamic Reconfigurations

As presented in Section 3.4.5.3 the life-cycle of each AWA activity is controlled by an *Autonomic Controller* (AC) based on a *State Machine*. In order to support dynamic reconfigurations this *State Machine* needs to be extended to fit the new requirements, namely the operation of the *Dynamic Reconfiguration Handler* (DRH) and the need for processing the reconfiguration operators. These requirements are covered by the new states and new state transitions depicted in Figure 4.12, enclosed within the dashed line and described in the following.

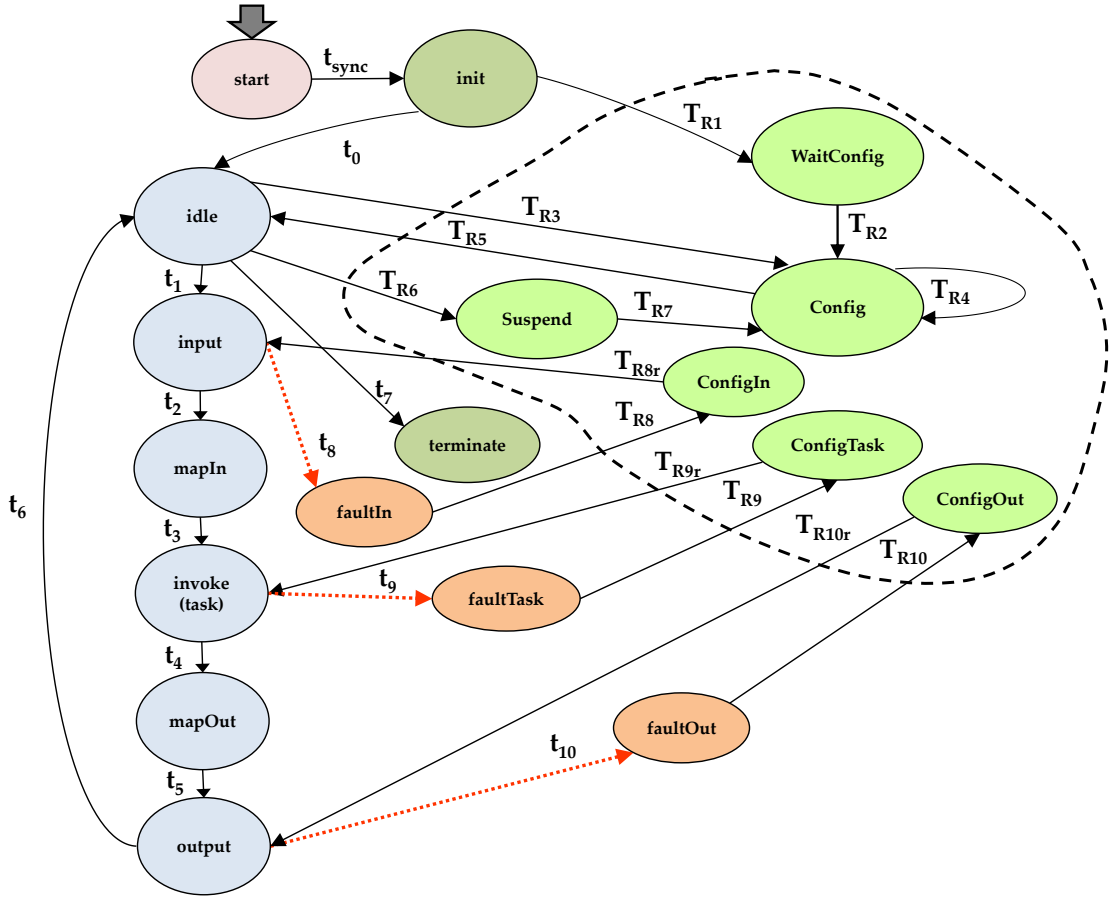


Figure 4.12: The State Machine extended to support dynamic reconfigurations

✧ **Initial state**

During the initial state (*init*) of the *State Machine* it is necessary to distinguish two cases.

The first case is when an activity is launched for running immediately and then the *State Machine* goes (transition t_0) to the *idle* state.

The second case is when a new activity is launched in the scope of a reconfiguration plan which requires the activity to wait for a reconfiguration block with the *StartExec* operator for allowing the activity to participate in the agreement that defines the earliest global iteration number K where the new activity will start its execution. This implies a

new $WaitConfig$ state in the *State Machine* to ensure that the new activity waits for the required reconfiguration block.

In order to distinguish between the two cases the specification of an activity needs to be changed to contain a new $InitialState$ item which can take the following two values: $idle$; or $WaitConfig$.

From the workflow example presented in Figure 4.9 on page 107 in Section 4.2.2, in Table 4.2 we illustrate the two cases.

The new *Feedback* activity launched during the reconfiguration plan specifies that after initialization its *State Machine* moves (T_{R1}) to the $WaitConfig$ state. Before applying the reconfiguration plan the workflow activity named *C* was launched for running immediately, so after initialization its *State Machine* moves (t_0) to the $idle$ state.

Table 4.2: Specification of activities with the *State Machine* initial state

AWA	*Name*	**Feedback**				
	InitialState	**WaitConfig**				
	Inputs	*Name*	*State*	*TokenType*	*IMode*	
		I1F	**Enable**	**java.lang.String[]**	**Iteration**	
	Outputs	*Name*	*State*	*TokenType*	*OMode*	*SendTo*
		O1F	**EnableFeedback**	**wkf.FilterCriteria**	**Single**	**I1A**
	Task	*Parameter*		**"server=S;database=DBHist;"**		
		SoftwareComponent		**wkf.TaskDataFiltering**		
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*	
				I1F	**0**	
		Mapping Results to Outputs		*Result Order*	*Output Name*	
				0	**O1F**	

AWA	*Name*	**C**			
	InitialState	**idle**			
	Inputs	*Name*	*State*	*TokenType*	*IMode*
		I1C	**Enable**	**wkf.Record**	**Iteration**
	Task	*Parameter*		**"server=S;database=DBREC;"**	
		SoftwareComponent		**wkf.TaskStoreRecords**	
		Mapping Inputs to Arguments		*Input Name*	*Argument Order*
				I1C	**0**
		Mapping Results to Outputs		*Result Order*	*Output Name*
				Null	**Null**

✧ **Moving to the *Config* state**

Before the beginning of any iteration in the $idle$ state the *State Machine* checks for still pending but already agreed upon reconfigurations at this iteration. If there is any pending reconfigurations the *State Machine* moves (T_{R3}) to the new $Config$ state where the operators of the corresponding reconfiguration block are processed (T_{R4}). Also, for supporting the $StartExec$ operator, when the *State Machine* is in the $WaitConfig$ state and the *Dynamic Reconfiguration Handler* (DRH) receives the agreement related to the needed reconfiguration block, the *State Machine* also moves (T_{R2}) to the new $Config$ state.

After processing a reconfiguration block ended by the $EndAwaReConfig$ operator in the $Config$ state, the next state of the *State Machine* always moves to the $idle$ state (T_{R5}).

However, for supporting possible recovery from faults by using dynamic reconfigurations, after the *Dynamic Reconfiguration Handler* (DRH) receives a dynamic reconfiguration bock the *State Machine* also moves by (T_{R8}), (T_{R9}) or (T_{R10}) respectively from the faulty states ($faultIn$, $faultTask$ or $faultOut$), to the corresponding fault recovery configuration states ($ConfigIn$, $ConfigTask$ or $ConfigOut$).

✧ **Moving to the *fault* recovery configuration states**

The reconfiguration plan used to recover an activity in the faulty states ($faultIn$, $faultTask$ or $faultOut$) should only include a reconfiguration block with the adequate operators for the activity.

As an example if the *State Machine* is in the $faultTask$ state the adequate reconfiguration actions are changing the activity $Task$ ($ChangeTask$ operator) or changing the activity $Parameters$ ($ChangeParameters$ operator).

When the *State Machine* is in the $faultIn$ or $faultOut$ faulty states, it can be more difficult to recover. In fact, failures related to the AWARD Space connectivity may cause difficulties for ensuring the atomicity for reading or writing all tokens of all input or output ports.

However, depending on the failure analysis and possible recovery actions, for instance the status replacement of the AWARD Space server, the workflow developer can submit reconfiguration plans for retrying the execution without faults.

The reconfiguration plans for recovering from failures must be issued for only one activity and must include the $RetryAfterFaultIn$, $RetryAfterFaultTask$ or $RetryAfterFaultOut$ operators for moving the *State Machine* from the corresponding fault recovery configuration states ($ConfigIn$, $ConfigTask$ or $ConfigOut$) back to the $input$, $invoke$ or $output$ state respectively (transitions T_{R8r}, T_{R9r} and T_{R10r} in Figure 4.12).

✧ **Moving to the *Suspend* state**

The new $Suspend$ state is related to the $Suspend/Resume$ operators. Before the beginning of any iteration in the $idle$ state the *State Machine* checks if a $Suspend$ operator was processed to be applied in this iteration and moves (T_{R6}) to the new $Suspend$ state.

Later, when a new reconfiguration block containing the *Resume* operator is processed the *State Machine* moves (T_{R7}) to the $Config$ state and after the processing of the $EndAwaReConfig$ operator, the *State Machine* comes back (T_{R5}) to the $idle$ state.

✧ **Redefinition of the *State Machine***

The above modification to the *State Machine* implies little changes to the definitions presented in Section 3.4.5.3. These changes are presented in the following:

Definition 4.8: *The new set of State Machine states:* S_R

According to Definition 3.23 (on page 81) the set (S) of states of the *State Machine*:

$$S = \{start, init, idle, input, mapIn, invoke, mapOut, output, terminate, faultIn, faultTask, faultOut\}$$

is redefined to the new set (S_R) of states:

$$S_R = S \cup \{WaitConfig, Config, Suspend, ConfigIn, ConfigTask, ConfigOut\}$$

where the new states and the actions performed in each state are described in Table 4.3 on page 118.

Definition 4.9: *The new set of state transitions:* T_R

According to Definitions 3.24 (on page 81) and 3.25 (on page 82), the set (T) of state transitions:

$$T = \{t_{sync}, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$$

is redefined to the new set (T_R) of transitions as:

$$T_R = T \cup \{T_{R1}, T_{R2}, T_{R3}, T_{R4}, T_{R5}, T_{R6}, T_{R7}, T_{R8}, T_{R9}, T_{R10}, T_{R8r}, T_{R9r}, T_{R10r}\}$$

where the t_0 and t_7 transitions are redefined in the new set (T_R). The new transitions including the redefinition of the t_0 and t_7 are described in Tables 4.4 (on page 118), 4.5 (on page 119), 4.6 (on page 119), 4.7 (on page 120) and 4.8 (on page 120).

Definition 4.10: ***The new global variables of the State Machine:*** V_R

According to Definition 3.26 (on page 83), the set (V) of global variables of the *State Machine* for defining the internal *AWA Context*:

$$V = \{IsToStart, CurIter, CurState, IsFaulty, InsReady, InsTokens, OutsTokens, TArgs, TRes\}$$

is extended to the new set (V_R):

$$V_R = V \cup \{WaitToConfig, EndAwaConfig, AgreedIter, isToSuspend, isToResume, isToTerminate\},$$

where the new global variables include information related to the new states and the associated new conditions, as described in Tables 4.9 (on page 120) and 4.10 (on page 120).

Definition 4.11: ***The new conditions of the*** **State Machine:** C_R

According to Definition 3.27 (on page 84), the set (C) of *State Machine* conditions:

$$C = \{c_{sync}, c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$$

is redefined to the new set (C_R):

$$C_R = C \cup \{C_{R0}, C_{R1}, C_{R2}, C_{R3}, C_{R4}, C_{R5}, C_{R6}, C_{R7}, C_{R8}, C_{R9}, C_{10}, C_{R8r}, C_{R9r}, C_{R10r}\}$$

Depending on the global variables presented in the above Definition 4.10, Table 4.10 (on page 120) presents the new conditions and the redefinition of the c_7 condition.

An activity launched in the scope of a reconfiguration plan starts its execution at the iteration agreed for applying the reconfiguration. Then the action A_{R2} associated to transition T_{R2} between states $WaitConfig$ and $Config$ (Table 4.5 on page 119) is made to assign the iteration agreed to the global variable which stores the current iteration of the activity and is defined as $A_{R2} = (CurrentIter \leftarrow AgreedIter)$ meaning that the activity starts the execution according to the iteration number agreed upon between all activities involved in the reconfiguration plan.

Table 4.3: The new states and actions of the *State Machine*

State	**Description**	**Actions of the *Autonomic Controller* (a_s)**
WaitConfig	When an activity is launched in the scope of a reconfiguration plan it needs to wait for a reconfiguration. Then, after the initialization, the *State Machine* moves to this state.	The *State Machine* waits in this state until the *Dynamic Reconfiguration Handler* handles an iteration agreement and then moves (T_{R2}) to the $Config$ state.
Suspend	At the beginning of any iteration, in the $idle$ state, if the activity was reconfigured to be suspended, the *State Machine* moves (T_{R6}) to the $Suspend$ state.	The *State Machine* waits in this state until the *Dynamic Reconfiguration Handler* handles a reconfiguration block with a *Resume* operator that resumes the execution of the activity.
Config	This is the state for processing reconfiguration blocks with sequences of operators handled by *Dynamic Reconfiguration Handler* (DRH). According to the operator semantics the *State Machine* produces facts and rules into the *Rules Engine*, for instance a fact with a new $Task$ name in a $ChangeTask$ operator or a fact with a new parameter list in a $ChangeParameters$ operator.	The *State Machine* processes the sequence of reconfiguration operators and according to the transitions for the $Config$ state, after processing the $EndAwaReConfig$ operator for ending the reconfiguration block, the *State Machine* moves to the $idle$ state (T_{R5}).
ConfigIn	This is the state for processing a reconfiguration block handled by *Dynamic Reconfiguration Handler* for trying recovering the activity failure on $input$ state.	The *State Machine* waits in this state until the *Dynamic Reconfiguration Handler* handles a reconfiguration block with a $RetryAfterFaultIn$ operator and then moves (T_{R8r}) to the $input$ state.
ConfigTask	This is the state for processing a reconfiguration block handled by *Dynamic Reconfiguration Handler* for trying recovering the activity failure on $invoke$ state.	The *State Machine* waits in this state until the *Dynamic Reconfiguration Handler* handles a reconfiguration block with a $RetryAfterFaultTask$ operator and then moves (T_{R9r}) to the $invoke$ state.
ConfigTask	This is the state for processing a reconfiguration block handled by *Dynamic Reconfiguration Handler* for trying recovering the activity failure on $output$ state.	The *State Machine* waits in this state until the *Dynamic Reconfiguration Handler* handles a reconfiguration block with a $RetryAfterFaultOut$ operator and then moves (T_{R10r}) to the $output$ state.

Table 4.4: The Init state transitions

Transition (t)	**Source State (s_i)**	**Destination State (s_j)**	**Condition (c)**	**Action (a_t)**
t_0 redefined	*init*	*idle*	c_{R0}: The specification field $initialState$ is $idle$, for running the activity immediately.	–
T_{R1}	*init*	*WaitConfig*	c_{R1}: The specification field $initialState$ is $WaitConfig$ for the activity waiting for a reconfiguration.	–

Table 4.5: Moving to/from the $Config$ state

Transition (t)	**Source State** (s_i)	**Destination State** (s_j)	**Condition** (c)	**Action** (a_t)
T_{R2}	*WaitConfig*	*Config*	c_{R2}: The *Dynamic Reconfiguration Handler* handles a reconfiguration agreement that must include the $StartExec$ operator.	A_{R2}: An activity launched starts its execution at the iteration number agreed
T_{R3}	*idle*	*Config*	c_{R3}: Current iteration is equal to an iteration agreed for applying a reconfiguration.	–
T_{R4}	*Config*	*Config*	c_{R4}: TRUE until processing the $EndAwaReConfig$ operator of the reconfiguration block.	–
T_{R5}	*Config*	*idle*	c_{R5}: TRUE if completed the processing of the $EndAwaReConfig$ operator of the reconfiguration block.	–

Table 4.6: Moving to/from fault reconfiguration states

Transition (t)	**Source State** (s_i)	**Destination State** (s_j)	**Condition** (c)	**Action** (a_t)
T_{R8}	*faultIn*	*ConfigIn*	c_{R8}: The *Dynamic Reconfiguration Handler* has handled a reconfiguration block that must include the $RetryAfterFaultIn$ operator for trying recovering the activity failure on $input$ state.	–
T_{R9}	*faultTask*	*ConfigTask*	c_{R9}: The *Dynamic Reconfiguration Handler* has handled a reconfiguration block that must include the $RetryAfterFaultTask$ operator for trying recovering the activity failure on $invoke$ state.	–
T_{R10}	*faultOut*	*ConfigOut*	c_{R10}: The *Dynamic Reconfiguration Handler* has handled a reconfiguration block that must include the $RetryAfterFaultOut$ operator for trying recovering the activity failure on $output$ state.	–
T_{R8r}	*ConfigIn*	*input*	c_{R8r}: TRUE if completed the processing of the $RetryAfterFaultIn$ operator to recover the activity failure on $input$ state.	–
T_{R9r}	*ConfigTask*	*invoke*	c_{R9r}: TRUE if completed the processing of the $RetryAfterFaultTask$ operator to recover the activity failure on $invoke$ state.	–
T_{R10r}	*ConfigOut*	*output*	c_{R10r}: TRUE if completed the processing of the $RetryAfterFaultOut$ operator to recover the activity failure on $output$ state.	–

Table 4.7: Moving to/from *Suspend* state

Transition (t)	**Source State** (s_i)	**Destination State** (s_j)	**Condition** (c)	**Action** (a_t)
T_{R6}	*idle*	*Suspend*	c_{R6}: Current iteration is equal to an iteration agreed for applying the reconfiguration *Suspend* operator.	–
T_{R7}	*Suspend*	*Config*	c_{R7}: Dynamic Reconfiguration Handler has handled a reconfiguration block with a *Resume* operator to resume the activity.	–

Table 4.8: Moving to *terminate* state

Transition (t)	**Source State** (s_i)	**Destination State** (s_j)	**Condition** (c)	**Action** (a_t)
t_7 redefined	*idle*	*terminate*	c_7: The current iteration reaches the maximum iteration number, or an explicit synchronous termination (*terminate* operator) occurred.	–

Table 4.9: Global variables and the associated conditions

Variable	**Type**	**Description**	**Related conditions**
$WaitToConfig$	*Boolean*	It indicates that the activity is launched in the scope of a reconfiguration plan and needs to wait for a reconfiguration block. The variable is related to the *initialState* field of the activity specification.	C_{R0}; C_{R1}
$EndAwaConfig$	*Boolean*	It indicates that the *Dynamic Reconfiguration Handler* completes a reconfiguration block by processing the $EndAwaReConfig$ operator.	c_7; C_{R2}; C_{R4}; C_{R5}; C_{R7}; C_{R8}; C_{R9}; C_{R10}; C_{R8r}; C_{R9r}; C_{R10r}
$AgreedIter$	*Integer*	Defines the iteration number agreed to perform a reconfiguration, indicating that the *Dynamic Reconfiguration Handler* already processed the compound $EndReConfiguration$ operator.	C_{R3}; C_{R6}
$isToSuspend$	*Boolean*	Indicates that the $Suspend$ operator was processed in the $Config$ state.	C_{R6}
$isToResume$	*Boolean*	Indicates that the *Resume* operator was processed in the $Config$ state.	C_{R7}
$isToTerminate$	*Boolean*	Indicates that the $Terminate$ operator was processed in the $Config$ state.	c_7

Table 4.10: The new conditions of the *State Machine*

$c_7(redefined) = (CurIter == MaxIter)\ or\ (EndAwaConfig\ and\ isToTerminate)$
$C_{R0} = not\ WaitToConfig$
$C_{R1} = WaitToConfig$
$C_{R2} = EndAwaConfig$
$C_{R3} = (CurIter == AgreedIter)$
$C_{R4} = not\ EndAwaConfig$
$C_{R5} = EndAwaConfig$
$C_{R6} = (CurIter == AgreedIter)\ and\ isToSuspend$
$C_{R7} = EndAwaConfig\ and\ isToResume$
$C_{R8}, C_{R9}, C_{R10} = isFaulty\ and\ EndAwaConfig$
$C_{R8r}, C_{R9r}, C_{R10r} = EndAwaConfig$

4.3.3 The Semantics of Operators as Transitions between AWA Contexts

Except for the compound operators used to define the scope of a reconfiguration plan and the activity reconfiguration blocks, the elementary dynamic reconfiguration operators presented in Section 4.2.1 are used for changing the context characteristics of the workflow activities involved in reconfiguration plans. These changes in the contexts of activities perform structural and behavioral modifications on the long-running workflow. Then considering an *AWA Context* the semantics of each dynamic reconfiguration operator can be described as a transition to a new *AWA Context*, where the new context reflects the modifications made by the operators.

Before presenting the context transitions caused by each operator let us review the definitions related to the context of an AWA activity. Recalling the Chapter 3 Definitions (3.17; 3.18; 3.19; 3.20; 3.21), and the new internal *AWA Context*, including the new global variables (as in Definition 4.10 on page 117), the complete context of the A activity, denoted by $Ctx(A)$, is defined as follows:

✧ **The Context of the *A* activity**

$Ctx(A) = \{CfCtx(A), IntCtx(A)\}$

✧ **Configuration Context of the *A* activity**

$CfCtx(A) = (APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$

✧ **Internal Context of the *A* activity**

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, CurState, IsFaulty,\\ &InsReady, InsTokens, OutsTokens, TArgs, TRes,\\ &WaitToConfig, EndAwaConfig, AgreedIter,\\ &isToSuspend, isToResume, isToTerminate) \end{aligned}$$

✧ **Configuration Context of the input ports of the *A* activity**

$CfCtx_{inputs} = \{CfCtx_{in}(I_1A), ..., CfCtx_{in}(I_nA)\}$

✧ **Configuration Context of an input port (I_iA) of the *A* activity**

$CfCtx_{in}(I_iA) = (I_iA, Ttype, IMode, State)$, $1 \leqslant i \leqslant n$

✧ **Configuration Context of the output ports of the *A* activity**

$CfCtx_{outputs} = \{CfCtx_{out}(O_1A), ..., CfCtx_{out}(O_nA)\}$

✧ **Configuration Context of an output port (O_iA) of the *A* activity**

$CfCtx_{out}(O_iA) = (O_iA, Ttype, OMode, SendTo, State)$, $1 \leqslant i \leqslant n$

✧ **Input ports mapping to a *Task* with *nargs Arguments***

$AMapins = \{(0, Iname_0), ..., (k, Iname_k), ..., (nargs-1, Iname_{nargs-1})\}$

where $0 \leqslant k \leqslant n\arg s - 1$ and $Iname_k \in A_{Inputs}$

✧ **Output ports mapping from the *res Task Results***

$AMapouts = \{(Oname_0, res_0), ..., (Oname_k, res_k), ..., (Oname_{n-1}, res_{n-1})\}$

where $0 \leqslant res_k \leqslant nres - 1$ and $Oname_k \in A_{Outputs}$

✧ **The semantics of the dynamic reconfiguration operators**

The semantics of a dynamic reconfiguration operator applied to an AWA activity (A) is defined as the $Ctx(A) \xrightarrow{Operator(arguments)} Ctx_{new}(A)$ transition from the *AWA Context* $Ctx(A)$ to a new *AWA Context* $Ctx_{new}(A)$. The transition is labelled with the operator name and the specific arguments of each operator.

In most operators only the activity configuration context $CfCtx(A)$ is modified. The internal context is modified in the operators related to the activity life-cycle, namely for starting, suspending, resuming or terminating an activity or when a new activity is launched. Therefore in the following operator transitions we only highlight the component of the activity context which is modified. A detailed description of the operator arguments is presented in Figure 5.13 on page 167.

Definition 4.12: *ChangeParameters*

Applying the *ChangeParameters* operator to the A activity changes the configuration context $CfCtx(A)$ by changing the current list of *Parameters* denoted by $APars$ to a new list of *Parameters* denoted by $AnewPars$.

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

$$\xrightarrow{\textbf{ChangeParameters}(\boldsymbol{AnewPars})}$$

$$(\boldsymbol{AnewPars}, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

Definition 4.13: *ChangeTask*

Applying the *ChangeTask* operator to the A activity changes the configuration context $CfCtx(A)$ by changing the current *Task* denoted by $ATask$ to a new task denoted by $AnewTask$.

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

$$\xrightarrow{\textbf{ChangeTask}(\boldsymbol{ANewTask})}$$

$$(APars, \boldsymbol{ANewTask}, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

Definition 4.14: *CreateInput*

Applying the $CreateInput$ operator to the A activity changes the configuration context $CfCtx(A)$ by changing the $CfCtx_{inputs}$ configuration context of the input ports to the new $NewCfCtx_{inputs}$ configuration context of the input ports, which includes, according to Definition 3.7 on page 60, the new $(InewNameA, Ttype, IMode, State)$ input port. Adding a new input port implies changing the $AMapins$ activity inputs mapping to the new $NewAMapins$ inputs mapping.

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

$$\xrightarrow{\textbf{CreateInput}(\textbf{\textit{InewNameA,Ttype,IMode,State}})}$$

$$(APars, ATask, \textbf{\textit{NewCfCtx}}_{\textbf{\textit{inputs}}}, \textbf{\textit{NewAMapins}}, CfCtx_{outputs}, AMapouts, MaxIter)$$

where, $\textbf{\textit{NewCfCtx}}_{\textbf{\textit{inputs}}} = CfCtx_{inputs} \bigcup CfCtx_{in}(\textbf{\textit{InewNameA}})$ is a new set of the input ports context, including the configuration context of the new input port named ***InewNameA***, $CfCtx_{in}(\textbf{\textit{InewNameA}}) = (\textbf{\textit{InewNameA,Ttype,IMode,State}})$;
The new ***NewAMapins*** inputs mapping function includes a new (***nargs,InewNameA***) pair to map the new input port to a new $Task$ argument:
$\textbf{\textit{NewAMapins}} = AMapins \bigcup \{(\textbf{\textit{nargs,InewNameA}})\}$ where $AMapins$ is the set $AMapins = \{(0, Iname_0), ..., (k, Iname_k), ..., (nargs-1, Iname_{nargs-1})\}$.

Definition 4.15: *ChangeInputOrder*

Applying the $ChangeInputOrder$ operator to an input port named $InameA$ of the A activity changes the $CfCtx(A)$ configuration context by changing the $IMode$ input order mode of the $CfCtx_{in}(InameA)$ configuration context of the $InameA$ input port to the new $InewMode \in \{Iteration, Sequence, Any\}$ input port mode.

$$CfCtx_{in}(InameA) = (InameA, Ttype, IMode, State)$$

$$\xrightarrow{\textbf{ChangeInputOrder}(\textbf{\textit{InewMode}})}$$

$$CfCtx_{in}(InameA) = (InameA, Ttype, \textbf{\textit{InewMode}}, State)$$

Note that if the $InewMode$ is the $Sequence$ input mode then this operation is only allowed if the A activity has a single input port connected to a simple link (Definition 3.7 on page 60)

Definition 4.16: *CreateOutput*

Applying the $CreateOutput$ operator to the A activity changes the configuration context $CfCtx(A)$ by changing the $CfCtx_{outputs}$ configuration context of the output ports to the new $NewCfCtx_{outputs}$ configuration context of the output ports, which includes, according to Definition 3.8 on page 60, the new $(OnewNameA, Ttype, OMode, SendTo, State)$ output port. Adding a new output port implies changing the $AMapouts$ activity outputs mapping to the new $NewAMapouts$ output mapping.

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

$$\xrightarrow{\textbf{CreateOutput}(\boldsymbol{OnewNameA,Ttype,OMode,SendTo,State})}$$

$$(APars, ATask, CfCtx_{inputs}, AMapins, \boldsymbol{NewCfCtx_{outputs}}, \boldsymbol{NewAMapouts}, MaxIter)$$

where, $\boldsymbol{NewCfCtx_{outputs}} = CfCtx_{outputs} \bigcup CfCtx_{out}(\boldsymbol{OnewNameA})$ is a new set of the output ports context including the configuration context of the new output port named $\boldsymbol{OnewNameA}$, $CfCtx_{out}(\boldsymbol{OnewNameA}) = (\boldsymbol{OnewNameA,Ttype,OMode,SendTo,State})$;
The new $\boldsymbol{NewAMapouts}$ outputs mapping function includes a new $(\boldsymbol{OnewNameA, res_n})$ pair to map a $Task$ result to the new output port:
$\boldsymbol{NewAMapouts} = AMapouts \bigcup \{(\boldsymbol{OnewNameA, res_n})\}$ where $AMapouts$ is the set $AMapouts = \{(Oname_0, res_0), ..., (Oname_k, res_k), ..., (Oname_{n-1}, res_{n-1})\}$.

Definition 4.17: *ChangeOutputLink*

Applying the $ChangeOutputLink$ operator to an output port named $OnameA$ of the A activity changes the $CfCtx(A)$ activity configuration context by changing the $CfCtx_{out}(OnameA)$ configuration context in order to change the link of the output port to a new destinations list.

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, SendTo, State)$$

$$\xrightarrow{\textbf{ChangeOutputLink}(\boldsymbol{NewSendTo})}$$

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, \boldsymbol{NewSendTo}, State)$$

Definition 4.18: *AddOutputLink*

Applying the $AddOutputLink$ operator to an output port named $OnameA$ of the A activity changes the $CfCtx(A)$ activity configuration context by changing the $CfCtx_{out}(OnameA)$ configuration context in order to change the link of the output port by adding the new $NewDest$ destination to the $SendTo$ destination list.

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, SendTo, State)$$

$$\xrightarrow{\textbf{AddOutputLink}(\boldsymbol{NewDest})}$$

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, \boldsymbol{NewSendTo}, State)$$

where $\boldsymbol{NewSendTo} = SendTo \cup \{\boldsymbol{NewDest}\}$

Definition 4.19: *ChangeOutputStrategy*

Applying the $ChangeOutputStrategy$ operator to an output port named $OnameA$ of the A activity changes the $CfCtx(A)$ activity configuration context by changing the $CfCtx_{out}(OnameA)$ configuration context in order to change the output port strategy, which consists of changing the $OMode$ current output mode to the new $OnewMode \in \{Single, Replicate, RoundRobin\}$ output mode.

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, SendTo, State)$$

$$\xrightarrow{\textbf{ChangeOutputStrategy}(\boldsymbol{OnewMode})}$$

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, \boldsymbol{OnewMode}, SendTo, State)$$

Definition 4.20: *ChangeInputState*

Applying the $ChangeInputState$ operator to an input port named $InameA$ of the A activity changes the $CfCtx(A)$ configuration context by changing the $CfCtx_{in}(InameA)$ input configuration context in order to change the current input port state ($State$) to the new $newState \in \{Enable, Disable, EnableFeedback\}$ input port state.

$$CfCtx_{in}(InameA) = (InameA, Ttype, IMode, State)$$

$$\xrightarrow{\textbf{ChangeInputState}(\boldsymbol{newState})}$$

$$CfCtx_{in}(InameA) = (InameA, Ttype, IMode, \boldsymbol{newState})$$

Definition 4.21: *ChangeOutputState*

Applying the $ChangeOutputState$ operator to an output port named $OnameA$ of the A activity changes the $CfCtx(A)$ activity configuration context by changing the $CfCtx_{out}(OnameA)$ output configuration context in order to change the current output port state ($State$) to the new $newState \in \{Enable, Disable, EnableFeedback\}$ output port state.

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, SendTo, State)$$

$$\xrightarrow{\textbf{ChangeOutputState}(\boldsymbol{newState})}$$

$$CfCtx_{out}(OnameA) = (OnameA, Ttype, OMode, SendTo, \boldsymbol{newState})$$

Definition 4.22: *ChangeMaxIterations*

Applying the $ChangeMaxIterations$ operator to the A activity changes the $CfCtx(A)$ configuration context by changing the $MaxIter$ maximum number of iterations to the new $NewMaxIter$ maximum number of iterations.

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$$

$$\xrightarrow{\textbf{ChangeMaxIterations}(\boldsymbol{NewMaxIter})}$$

$$(APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, \boldsymbol{NewMaxIter})$$

Definition 4.23: *ChangeMappingInputs*

Applying the $ChangeMappingInputs$ operator to the A activity changes the $CfCtx(A)$ configuration context by changing the $AMapins$ map function of input ports to the $Task\ Arguments$ to the new $NewAMapins$ map function.

$$AMapins = \{(0, Iname_0), ..., (k, Iname_k), ..., (nargs-1, Iname_{nargs-1})\}$$

$$\xrightarrow{\textbf{ChangeMappingInputs}(\boldsymbol{NewAMapins})}$$

$$\boldsymbol{NewAMapins} = \{(0, \boldsymbol{InewName_0}), ..., (k, \boldsymbol{InewName_k}), ..., (nargs-1, \boldsymbol{InewName_{nargs\text{-}1}})\}$$

where $\boldsymbol{NewAMapins}$ maps the input ports to $Task\ Arguments$ in other arbitrary order, as the set, $\{(0, \boldsymbol{InewName_0}), ..., (k, \boldsymbol{InewName_k}), ..., (nargs-1, \boldsymbol{InewName_{nargs\text{-}1}})\}$

Definition 4.24: *ChangeMappingOutputs*

Applying the $ChangeMappingOutputs$ operator to the A activity changes the $CfCtx(A)$ configuration context by changing the $AMapouts$ map function of $Task\ Results$ to output ports to the new $NewAMapouts$ map function.

$$AMapouts = \{(Oname_0, res_0), ..., (Oname_k, res_k), ..., (Oname_{n-1}, res_{n-1})\}$$

$$\xrightarrow{\textbf{ChangeMappingOutputs}(\boldsymbol{NewAMapouts})}$$

$$\boldsymbol{NewAMapouts} = \{(\boldsymbol{OnewName_0}, res_0), ..., (\boldsymbol{OnewName_k}, res_k), ..., (\boldsymbol{OnewName_{n\text{-}1}}, res_{n-1})\}$$

where ***NewAMapouts*** maps the $Task\ Results$ to output ports in other arbitrary order, as the set, $\{(\boldsymbol{OnewName_0}, res_0), ..., (\boldsymbol{OnewName_k}, res_k), ..., (\boldsymbol{OnewName_{n\text{-}1}}, res_{n-1})\}$

✧ **Life-Cycle operators**

The life-cycle operators only change the activity internal context ($IntCtx(A)$).

Definition 4.25: *StartExec*

The $StartExec$ operator is used only to reconfigure activities launched in the scope of a reconfiguration plan in order to notify the *State Machine* that it must move to the $Config$ state. This operator also defines the iteration where the activity must start according to the agreement achieved in the reconfiguration plan. Then applying the $StartExec$ operator to the A activity only implies changing the $IntCtx(A)$ internal context by changing the $curState = WaitConfig$ current state of the *State Machine* to the new $NewcurState = Config$ state, and the $CurIter$ current iteration number to the iteration agreed ($AgreedIter$).

$$IntCtx(A) = (IsToStart, \boldsymbol{CurIter=0}, \boldsymbol{CurState=WaitConfig}, IsFaulty, InsReady, InsTokens, OutsTokens, TArgs, TRes, WaitToConfig, EndAwaConfig, AgreedIter, isToSuspend, isToResume, isToTerminate)$$

$$\xrightarrow{\textbf{StartExec}()}$$

$$IntCtx(A) = (IsToStart, \boldsymbol{CurIter=AgreedIter}, \boldsymbol{CurState=Config}, IsFaulty, InsReady, InsTokens, OutsTokens, TArgs, TRes, \boldsymbol{WaitToConfig=FALSE}, EndAwaConfig, AgreedIter, isToSuspend, isToResume, isToTerminate)$$

Definition 4.26: *Suspend*

Applying the *Suspend* operator to the A activity only changes the $IntCtx(A)$ internal context by changing the $CurState = idle$ current state of the *State Machine* to the new $CurState = Suspend$ state.

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{CurState=idle}, IsFaulty,\\ &InsReady, InsTokens, OutsTokens, TArgs, TRes,\\ &WaitToConfig, EndAwaConfig, AgreedIter,\\ &\textbf{isToSuspend=TRUE}, isToResume, isToTerminate) \end{aligned}$$

$$\xrightarrow{\textbf{Suspend}()}$$

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{CurState=Suspend}, IsFaulty,\\ &InsReady, InsTokens, OutsTokens, TArgs, TRes,\\ &WaitToConfig, EndAwaConfig, AgreedIter,\\ &\textbf{isToSuspend=FALSE}, isToResume, isToTerminate) \end{aligned}$$

Definition 4.27: *Resume*

Applying the *Resume* operator to the A activity only changes the $IntCtx(A)$ internal context by changing the $CurState = Suspend$ current state of the *State Machine* to the new $CurState = Config$ state.

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{CurState=Suspend}, IsFaulty,\\ &InsReady, InsTokens, OutsTokens, TArgs, TRes,\\ &WaitToConfig, EndAwaConfig, AgreedIter,\\ &isToSuspend, \textbf{isToResume=TRUE}, isToTerminate) \end{aligned}$$

$$\xrightarrow{\textbf{Resume}()}$$

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{CurState=Config}, IsFaulty,\\ &InsReady, InsTokens, OutsTokens, TArgs, TRes,\\ &WaitToConfig, EndAwaConfig, AgreedIter,\\ &isToSuspend, \textbf{isToResume=FALSE}, isToTerminate) \end{aligned}$$

Definition 4.28: *Terminate*

Applying the $Terminate$ operator to the A activity only changes the $IntCtx(A)$ internal context by changing the $CurState = idle$ current state of the *State Machine* to the new $CurState = terminate$ state.

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{\textit{CurState=idle}}, IsFaulty, \\ &InsReady, InsTokens, OutsTokens, TArgs, TRes, \\ &WaitToConfig, EndAwaConfig, AgreedIter, \\ &isToSuspend, isToResume, \textbf{\textit{isToTerminate=TRUE}}) \end{aligned}$$

$$\xrightarrow{\textbf{Terminate}()}$$

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{\textit{CurState=terminate}}, IsFaulty, \\ &InsReady, InsTokens, OutsTokens, TArgs, TRes, \\ &WaitToConfig, EndAwaConfig, AgreedIter, \\ &isToSuspend, isToResume, \textbf{\textit{isToTerminate=FALSE}}) \end{aligned}$$

Definition 4.29: *RetryAfterFaultIn, RetryAfterFaultTask, RetryAfterFaultOut*

Applying the $RetryAfterFaultIn$, $RetryAfterFaultTask$ and $RetryAfterFaultOut$ operators for retrying fault recovery of the A activity only changes the $IntCtx(A)$ internal context by changing the $CurState \in \{ConfigIn, ConfigTask, ConfigOut\}$ current state of the *State Machine* to the new $CurState \in \{input, invoke, output\}$ state according respectively to the occurred $\{faultIn, faultTask, faultOut\}$ fault states

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{\textit{CurState}} \in \{\textbf{\textit{ConfigIn,ConfigTask,ConfigOut}}\}, \\ &\textbf{\textit{IsFaulty=TRUE}}, InsReady, InsTokens, OutsTokens, TArgs, TRes, \\ &WaitToConfig, EndAwaConfig, AgreedIter, \\ &isToSuspend, isToResume, isToTerminate) \end{aligned}$$

$$\xrightarrow{\{\textbf{RetryAfterFaultIn, RetryAfterFaultTask, RetryAfterFaultOut}\}()}$$

$$\begin{aligned} IntCtx(A) = (&IsToStart, CurIter, \textbf{\textit{CurState}} \in \{\textbf{\textit{input,invoke,output}}\}, \\ &\textbf{\textit{IsFaulty=FALSE}}, InsReady, InsTokens, OutsTokens, TArgs, TRes, \\ &WaitToConfig, EndAwaConfig, AgreedIter, \\ &isToSuspend, isToResume, isToTerminate) \end{aligned}$$

Definition 4.30: *LaunchActivity*

Applying the $LaunchActivity$ operator to launch an activity named A initializes the configuration context ($CfCtx(A)$) of the activity from the activity specification file ($specificationFile$) as well as the internal context ($IntCtx(A)$ with the $WaitConfig$ state as the current state of the *State Machine*.

$$Ctx(A) = \{-, -\}$$

$$\xrightarrow{\textbf{LaunchActivity}(\textbf{\textit{A, specificationFile}})}$$

$$Ctx(A) = \{\textbf{\textit{CfCtx(A)}}, \textbf{\textit{IntCtx(A)}}\}$$

Where null contexts are denoted by $(-)$ and the configuration context of the new A activity, denoted by ***CfCtx(A)*** $= (APars, ATask, CfCtx_{inputs}, AMapins, CfCtx_{outputs}, AMapouts, MaxIter)$ is initialized from the information in the specification file named ***specificationFile*** and the ***IntCtx(A)*** activity internal context reflects the initialization of the *State Machine* as follows:

$IntCtx(A) =$ (***IsToStart=TRUE, CurIter=0, CurState=WaitConfig, IsFaulty=FALSE, InsReady=FALSE, InsTokens=Null, OutsTokens=Null, TArgs=Null, TRes=Null, WaitToConfig=TRUE, EndAwaConfig=FALSE, AgreedIter=-1, isToSuspend=FALSE, isToResume=FALSE, isToTerminate=FALSE***)

Definition 4.31: *GetAwaContext*

Applying the $GetAwaContext$ operator to the A activity returns the $Ctx(A)$ activity context. The operator does not change the configuration and internal activity contexts.

4.4 Scenarios for Dynamic Reconfigurations

The requirements of a scientific application may not be completely defined at the beginning of an experiment thus forcing scientists to discuss and decide on taking subsequent actions, which can be based on intermediate experimental results, or/and based on newly gathered advice by other expert scientists. If the experiments are based on workflows, scientists can decide to change the workflows in order to achieve their goals. However, if a workflow is already running it can be costly, especially concerning wasting processing time, to stop the execution of the entire workflow for restarting a new workflow which includes the needed modifications. Therefore the workflow execution environment must support dynamic reconfigurations of long-running workflows according to the application demands, for example for enhancing application performance or improving its functionality.

For example, scientists should be able to dynamically change the workflow activity *Tasks*, the activity *Parameters* or even change the workflow structure by introducing new activities for instance for performing data filtering. Furthermore in scenarios where workflow activities are independently running on distributed infrastructures with multiple sites the workflow activities should be independently monitored and independently steered using dynamic reconfigurations by multiple users.

As presented in Section 4.2 the AWARD model provides a set of operators to elaborate scripts containing reconfiguration plans used to apply dynamic structural and behavioral modifications to long-running workflows.

Depending on each application scenario when a workflow is running there are multiple possible reasons and consequently multiple reconfiguration plans that can be applied to dynamically change the workflow. These reasons can result from some monitoring process allowing observing intermediate results, thus detecting activity failures, or raising the need to reduce the execution time of some activities, or the need to extend the workflow functionality. Therefore for each workflow the universe of all possible workflow reconfiguration scenarios is unlimited, because the possible reconfigurations have dependencies on the problem domain, as well as dependencies on events that happen during the execution of the workflow and that are often hard to predict before execution, that is at the workflow design time.

As an example, let us consider a simple workflow as depicted in Figure 4.13, where several possible reconfiguration plans can be applied according to the specific goals of each concrete application scenario. In a first scenario, reasons related to the inappropriate behavior of the *A*, *B* and *C* activities can lead the workflow developer to decide changing the *Tasks* of the activities and/or their *Parameters*. Then multiple and distinct reconfigurations plans can be applied, such as change the *Task* and/or *Parameters* of each activity separately, or even change the *Tasks* and/or their *Parameters* of the three activities jointly in a unique reconfiguration plan.

In other scenarios, due to reasons related to the structure of the workflow and the need

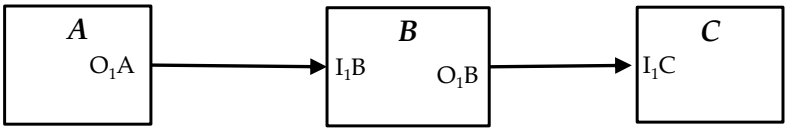


Figure 4.13: A simple long-running workflow

for monitoring of intermediate results, the workflow developer can decide to introduce new activities to perform new processing steps, such as introducing a filtering or data transformation activity between *A* and *B*, or between *B* and *C*, or even introducing two new distinct activities, one between *A* and *B* and a second between *B* and *C*. Furthermore the workflow developer can decide to introduce a new activity to introduce a feedback loop to be established between *B* and *A* or between *C* and *A*, which involves the creation of a new input port on the *A* activity and a new output port respectively on *B* or *C* activities.

Given the multitude of possible scenarios the approach followed in this dissertation consisted of finding a set of useful and realistic scenarios capable of addressing the following goals:

1. To validate and improve the design and implementation of a basic set of dynamic workflow reconfiguration operators;

2. To evaluate concrete cases of applications based on workflows that reveal possible scenarios for applying reconfiguration plans involving the reconfiguration of multiple workflow activities using the proposed dynamic reconfiguration operators;

3. To exercise and evaluate the developed AWARD prototype by implementing several application examples, as discussed in Chapter 6.

The next sections describe several selected useful workflow reconfiguration scenarios, including the scripts of each reconfiguration plan based on the skeleton described in Section 4.2 (Listing 4.1 on page 103). These reconfiguration plan scripts also illustrate the invocation details of each dynamic reconfiguration operator.

4.4.1 Scenario 1: Change the Task and the Parameters of Activities

As depicted in Figure 4.14 let us consider a workflow based on the *Synchronization AND-join* pattern [Aal+00b] where the *A* and *B* activities are executed in parallel and the *C* activity synchronously joins the tokens produced by the *A* and *B* activities. The last activity, named *D*, is used to output the workflow result and is only executed after the execution of the *C* activity, as in the *Sequence* pattern [Aal+00b].

In order to illustrate a dynamic reconfiguration scenario let us consider the workflow in Figure 4.14 as a long-running workflow with multiple iterations and assume that, during its execution it requires dynamic reconfigurations, for example for changing the activity *Tasks*, and/or for changing the *Parameters* of the activities. This can be used for improving the application performance or changing the functionality of activities.

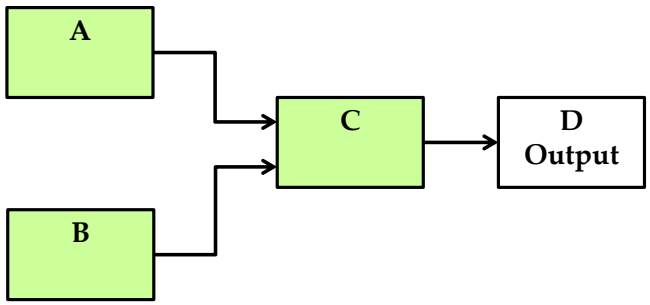


Figure 4.14: Workflow based on *Synchronization AND-join* pattern

Assuming that there is a software library (named *scenariostasks*) containing multiple implementations of the activity *Tasks* a possible reconfiguration plan is presented in Listing 4.5. The scope of the reconfiguration plan involves the *A*, *B* and *C* activities in order to change the *Tasks* of the *A* and *C* activities respectively to *scenariotasks.TaskAnew* and *scenariotasks.TaskCnew* and to change the *Parameters* of the *B* activity to the new *Parameters* list {*"TaskB","p1","p2","pnew"*}.

Listing 4.5: Reconfiguration plan to change activity *Tasks* and activity *Parameters*

```
int RID = BeginReConfiguration("A","C","B");
   BeginAwaReConfig(RID, "A");
     ChangeTask(RID, "A", "scenariotasks.TaskAnew");
   int it1 = EndAwaReConfig(RID, "A");
   BeginAwaReConfig(RID, "C");
     ChangeTask(RID, "C", "scenariotasks.TaskCnew");
   int it2 = EndAwaReConfig(RID, "C");
   BeginAwaReConfig(RID, "B");
     ChangeParameters(RID, "B", new String[]{"TaskB", "p1", "p2", "pnew"});
   int it3 = EndAwaReConfig(RID, "B");
   int[] AgreementSet=new int[]{it1, it2, it3};
int K=EndReConfiguration(RID, AgreementSet);
```

It is important to recall that the AWARD machine supports the submission of multiple concurrent reconfiguration plans with distinct reconfiguration identifiers (*RID*) meaning that this script can be submitted multiple times with different *Tasks* to *A* and *C* activities and different *Parameters* for the *B* activity. For instance, as presented in Listing 4.6 a different reconfiguration plan can be submitted later for changing the *Task* of the *B* activity to a new *Task* named *scenariotasks.TaskBnew*.

Listing 4.6: Reconfiguration plan to only change the *Task* of the *B* activity

```
int RID = BeginReConfiguration("B");
   BeginAwaReConfig(RID, "B");
     ChangeTask(RID, "B", "scenariotasks.TaskBnew");
   int it1 = EndAwaReConfig(RID, "B");
   int[] AgreementSet=new int[]{it1};
int K=EndReConfiguration(RID, AgreementSet);
```

The value of *K* in the above reconfiguration plans indicates the agreed upon iteration number such that the reconfiguration plan takes effect for producing the new workflow configuration to be applied at iteration *K*.

4.4.2 Scenario 2: Introduce New Activities for Monitoring Workflow Executions

During the execution of long-running workflows the analysis of intermediate results, for instance for verifying some behavior expectations, is mandatory. Typically in some workflow systems this is only supported by logging mechanisms that must be planned and considered at the workflow design time. The support provided by the AWARD model to dynamically reconfigure workflows introduces great flexibility for monitoring workflows without the need of considering this at workflow design time.

The scenario presented in Figure 4.15 considers the same workflow pattern as in scenario 1 (Figure 4.14), but now we consider that during the workflow execution at a certain point the workflow developer needs to monitor and store the data flowing between the *A* and *C* activities for future analysis.

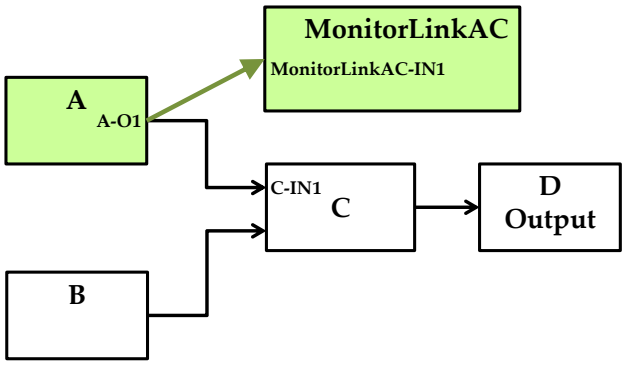


Figure 4.15: Monitoring the link between the *A* and *C* activities

Among other possibilities, for instance to change the *Task* of the *A* activity for doing some type of logging, a more flexible approach consists of dynamically launching a new workflow activity, named *MonitorLinkAC*, for monitoring the link between the *A* and *C* activities.

In order to perform the workflow reconfiguration required for monitoring the link between the *A* and *C* activities, as shown in Figure 4.15, the reconfiguration plan needs to launch the new *MonitorLinkAC* activity specified in the *MonitorLinkAC.xml* file. Also the *A-O1* output port of the *A* activity changes to support a multi-link by changing its destination list to include the *MonitorLinkAC-IN1* input port of the new activity.

The scope of the reconfiguration plan involves the *A* and *MonitorLinkAC* activities.

The script of the reconfiguration plan is presented in Listing 4.7, where the reconfiguration operator (*ChangeOutputLink*) changes the destination list of the *A-O1* output port

to include the *MonitorLinkAC-IN1* input port in addition to the *C-IN1* input port of the *C* activity. Also the output mode strategy of the *A-O1* output port is changed to *Replicate*.

Listing 4.7: Reconfiguration plan for monitoring the link between the *A* and *C* activities

```
int RID = BeginReConfiguration("MonitorLinkAC", "A");
  LaunchActivity("MonitorLinkAC.xml", "MonitorLinkAC");
  BeginAwaReConfig(RID, "MonitorLinkAC");
    StartExec(RID,"MonitorLinkAC");
  int it1 = EndAwaReConfig(RID,"MonitorLinkAC");
  BeginAwaReConfig(RID, "A");
    ChangeOutputLink(RID,"A","A-O1",new String[]{"C-IN1","MonitorLinkAC-IN1"});
    ChangeOutputStrategy(RID, "A", "A-O1", "Replicate");
  int it2 = EndAwaReConfig(RID,"A");
  int[] AgreementSet=new int[]{it1,it2};
int K=EndReConfiguration(RID, AgreementSet);
```

Another approach for monitoring a workflow activity is depicted in Figure 4.16. The reconfiguration plan, whose script is presented in Listing 4.8, consists of changing the *Task* of the *C* activity to produce data to be sent to a new output port, named *C-O2*, connected to the *CollectC-IN1* input port of the new *CollectC* activity. The scope of the reconfiguration plan involves the *C* activity and the new *CollectC* activity.

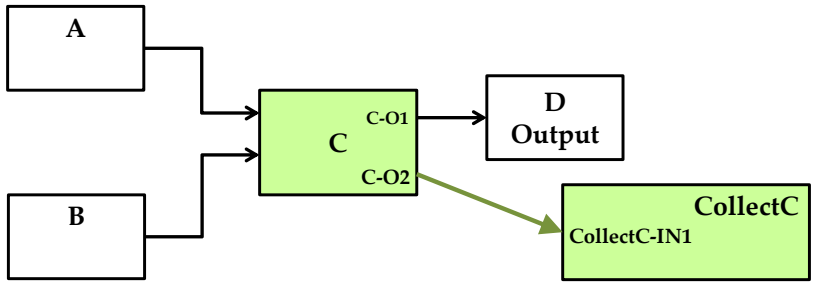


Figure 4.16: Change the activity *C* for collecting data

Listing 4.8: Reconfiguration plan for collecting data from the *C* activity

```
int RID = BeginReConfiguration("C", "CollectC");
  BeginAwaReConfig(RID, "C");
    ChangeTask(RID, "C", "scenariotasks.TaskCwith2outputs");
    CreateOutput(
      RID,"C","C-O2","TypeToken","Single",new String[]{"CollectC-IN1"},"Enable"
    );
    ChangeMappingOutputs(RID, "C", new String[]{"C-O1","C-O2"});
  int it1 = EndAwaReConfig(RID, "C");
  LaunchActivity("CollectC.xml", "CollectC");
  BeginAwaReConfig(RID, "CollectC");
      StartExec(RID, "CollectC");
  int it2 = EndAwaReConfig(RID, "CollectC");
  int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID, AgreementSet);
```

The script of the reconfiguration plan presented in Listing 4.8 shows the use of the *CreateOutput* operator whose arguments are according to the output port configuration context (Definition 3.8 on page 60: $CfCtx_{out} = (OnameA, Ttype, OMode, SendTo, State)$).

The script also shows the use of the *ChangeMappingOutputs* operator to change mappings of the *C-O1* and *C-O2* output ports according to the new *Task Results*.

4.4.3 Scenario 3: Change the Workflow Structure

The introduction of new activities can be useful for modifying the workflow structure in multiple ways. Let us consider a workflow based on the *Sequence* pattern [Aal+00b] which is a three steps pipeline with three activities named *Fa*, *Fb*, and *Fc* as depicted in Figure 4.17.

We assume that, after a certain time, by monitoring the execution behavior of the workflow, the workflow developer concludes that it is possible to improve the expected results of the workflow by changing the workflow structure as follows.

The output data of the *A* activity is also processed in parallel by a new activity, named *Fn*, with a *Task* which implements a new algorithm.

The *Task* of the *Fc* activity is changed to receive the output data of two activities: The existing *Fb* activity; and the output data from the new *Fn* activity.

In this way the *Fc* activity can choose the best result among the data produced by two distinct algorithms.

The scope of the reconfiguration plan involves the *Fa*, *Fc* and *Fn* activities.

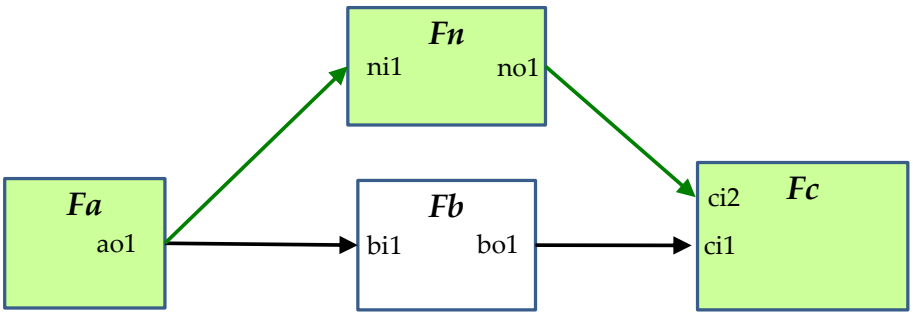


Figure 4.17: Change the structure of a workflow with distinct processing

The script of the reconfiguration plan is presented in Listing 4.9 showing the use of the following two operators: The *AddOutputLink* operator, to add an additional link which is the *ni*1 input port of the new *Fn* activity to the *ao*1 output port of the *Fa* activity; and the *CreateInput* operator whose arguments are according to the input port configuration context (Definition 3.7 on page 60: $CfCtx_{in} = (InameA, Ttype, IMode, State)$) to add the new *ci*2 output port to the *Fc* activity.

Listing 4.9: Reconfiguration plan to change the structure of a workflow

```
int RID = BeginReConfiguration("Fn", "Fa", "Fc");
   LaunchActivity("FnSpec.xml", "Fn");
   BeginAwaReConfig(RID, "Fn"););
        StartExec(RID, "Fn");
        ChangeOutputLink(RID, "Fn", "no1", new String[]{"ci2"});
   int it1 = EndAwaReConfig(RID, "Fn");;
   BeginAwaReConfig(RID, "Fa");
        AddOutputLink(RID, "Fa","ao1", new String[]{"ni1"});
        ChangeOutputStrategy(RID, "Fa", "ao1", "Replicate");
   int it2 = EndAwaReConfig(RID, "Fa");
   BeginAwaReConfig(RID, "Fc");
        CreateInput(RID, "Fc","ci2", "TypeToken", "Iteration", "Enable");
        ChangeTask(RID, "Fc", "scenariotasks.TaskFcWith2Args");
        ChangeMappingInputs(RID, "Fc", new String[]{"ci2", "ci1"});
   int it3 = EndAwaReConfig(RID, "Fc");
   int[] AgreementSet=new int[]{it1, it2, it3};
int K=EndReConfiguration(RID, AgreementSet);
```

Another useful scenario to dynamically change the structure of a workflow is to introduce a new activity for the purposes of data filtering or even data translation as depicted in Figure 4.18.

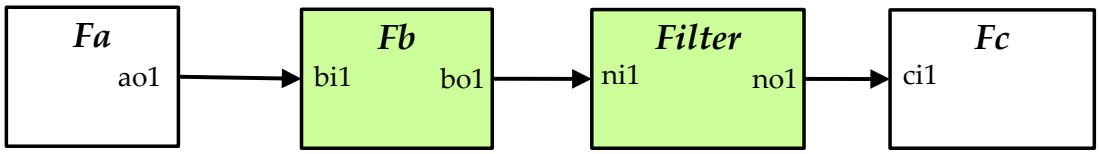


Figure 4.18: Change the workflow structure for filtering or translating data

In this scenario the pipeline workflow with *Fa*, *Fb* and *Fc* activities is modified to introduce the new *Filter* activity between the *Fb* and *Fc* activities.

The required reconfiguration plan is simple and consists of launching the new *Filter* activity and of changing the link of the *bo*1 output port of the *Fb* activity to be connected to the *ni*1 input port of the new *Filter* activity.

The scope of the reconfiguration plan involves the *Fb* and the new *Filter* activities.

The script of the reconfiguration plan is presented in Listing 4.10. All reconfiguration operators involved have been used in the previous scenarios.

It is important to note that the *Fc* activity is not involved in the reconfiguration plan. In fact, the semantics associated with input ports ensures that the *Fc* activity consumes tokens sent from the *Fb* activity until the $(K-1)^{th}$ iteration. At the K^{th} iteration, as agreed between *Fb* and *Filter* activities for applying the reconfiguration plan, the *Fb* activity starts sending tokens to the new *Filter* activity, which produces tokens that will be consumed by the *Fc* activity.

Listing 4.10: Reconfiguration plan for filtering or translate data

```
int RID = BeginReConfiguration("Filter", "Fb",);
   LaunchActivity("FilterSpec.xml", "Filter");
   BeginAwaReConfig(RID, "Filter"););
        StartExec(RID, "Filter");
        ChangeOutputLink(RID, "Filter", "no1", new String[]{"ci1"});
   int it1 = EndAwaReConfig(RID, "Filter");;
   BeginAwaReConfig(RID, "Fb");
        ChangeOutputLink(RID, "Fb", "bo1", new String[]{"ni1"});
   int it2 = EndAwaReConfig(RID, "Fb");
   int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID, AgreementSet);
```

4.4.4 Scenario 4: Change the Workflow Structure with a Feedback Loop

Although many workflow systems only allow direct acyclic graphs (DAG) thus limiting their use only for workflows without loops, the AWARD model allows the design of workflows with loops. Furthermore the AWARD support for dynamic reconfigurations of long-running workflows allows introducing these loops during the execution of a workflow.

To illustrate this scenario let us consider a pipeline workflow with the *Fa*, *Fb* and *Fc* activities as presented in Figure 4.19.

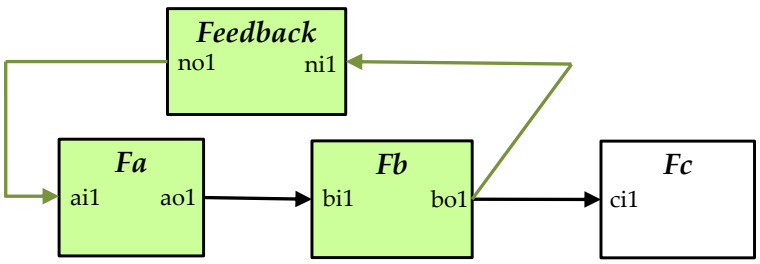


Figure 4.19: Change the workflow structure for introducing a feedback loop

Assume that during the execution of the workflow we can improve the workflow results by changing the workflow structure with a feedback loop for improving the *Fa* activity with information produced by the *Fb* activity in a previous iteration. Due to the sequence pattern of the workflow the feedback loop implies that tokens produced by the *bo*1 output port and processed by the new *Feedback* activity at the K^{th} iteration are processed by the new *ai*1 input port of the *Fa* activity at the $(K+1)^{th}$ iteration.

Therefore the *no*1 output port of the new *Feedback* activity and the new *ai*1 input port of the *Fa* activity are configured to the *EnableFeedback* state (Definition 3.15 on page 64). The *Task* of the *Fa* activity also needs to be changed in order to process the new argument which results from the input mapping of the new *ai*1 input port. For the *Fb* activity it is only necessary to change the destination links of the *bo*1 output port and change the output mode strategy to *Replicate*.

The scope of the reconfiguration plan involves the *Fa*, *Fb* and *Feedback* activities.

The script of the reconfiguration plan is presented in Listing 4.11. To illustrate the use of dynamic operators the reconfiguration block of the *Feedback* activity includes the *ChangeOutputLink*, *ChangeOutputState* and *ChangeOutputStrategy* operators to reconfigure the *no*1 output port. However, these operators can be omitted if these required output port configurations have been defined in the initial specification of the *Feedback* activity in the *FeedbackSpec.xml* file.

Listing 4.11: Reconfiguration plan to introduce a feedback loop

```
int RID = BeginReConfiguration("Fa","Fb","Feedback");
    BeginAwaReConfig(RID,"Fa");
       CreateInput(RID,"Fa","ai1","TypeToken","Iteration","EnableFeedback");
       ChangeTask(RID,"Fa","scenariotasks.TaskFaWithOneArg");
       ChangeMappingInputs(RID,"Fa",new String[]{"ai1"});
    int it1 = EndAwaReConfig(RID,"Fa");
    BeginAwaReConfig(RID,"Fb");
       AddOutputLink(RID,"Fb","bo1", new String[]{"ni1"});
       ChangeOutputStrategy(RID,"Fb","bo1","Replicate");
    int it2 = EndAwaReConfig(RID,"Fb");
    LaunchActivity("FeedbackSpec.xml","Feedback");
    BeginAwaReConfig(RID,"Feedback");
       StartExec(RID,"Feedback");
       ChangeOutputLink(RID,"Feedback","no1",new String[]{"ai1"});
       ChangeOutputState(RID,"Feedback", "no1","EnableFeedback");
       ChangeOutputStrategy(RID,"Feedback","no1","Single");
    int it3 = EndAwaReConfig(RID,"Feedback");
    int[] AgreementSet=new int[]{it1, it2, it3};
int K=EndReConfiguration(RID, AgreementSet);
```

4.4.5 Scenario 5: Introduce New Activities for Load Balancing

Independently of the workflow topology and the number of activities there are always workflow segments where some activities exhibit sequential dependencies with respect to other activities. If an activity has a slow pace because its *Task* exhibits a high execution time for each token processed then all successor activities are affected and will be left waiting for tokens produced by the slower activity.

As an example let us consider the workflow based on a sequence pattern formed by the *Fa*, *Fb* and *Fc* activities presented in Figure 4.20.

Let us assume that the *Task* of the *Fa* activity reads data items from files; the *Task* of the *Fb* activity processes the data items, and the *Task* of the *Fc* activity stores the results into a database. Assuming that the *Fb* activity has a slow pace, an adequate solution is to use a load balancing scenario where the data items produced by *Fa* are distributed to multiple replicas of the *Fb* activity for alternatively processing them in parallel.

The AWARD support for dynamic reconfigurations allows introducing new activities for load balancing purposes during a long-running workflow. Without the need to restart

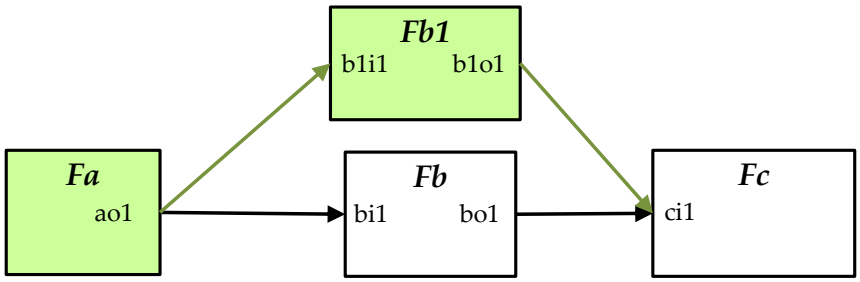


Figure 4.20: Change the workflow for load balancing of the *Fb* activity

the workflow execution anew, as depicted in Figure 4.20, a reconfiguration plan can launch a replica of the *Fb* activity, hereby named *Fb*1, for processing tokens alternatively in a *round-robin* fashion.

The reconfiguration plan also needs to change the mode of the *ao*1 output port of the *Fa* activity to the *RoundRobin* mode (Definition 3.10 on page 61) by using the reconfiguration *ChangeOutputStrategy* operator and also change the *bi*1 and *b*1*i*1 input ports, respectively, of the *Fb* and the *Fb*1 activities to the *Sequence* input mode (Definition 3.7 on page 60) by using the reconfiguration *ChangeInputOrder* operator.

The scope of the reconfiguration plan involves the *Fa*, *Fb* and *Fb*1 activities.

The script of the reconfiguration plan is presented in Listing 4.12 where the *Fc* activity is not involved in the reconfiguration plan because on each iteration, the *ci*1 input port continues receiving tokens still using the *Iteration* input mode without the need to know if their origin is from the *Fb* or from the *Fb*1 activities. According to Definition 3.4 on page 58 the tokens received by the *ci*1 input port follow the iteration numbers marked by the upstream *Fa* activity.

Listing 4.12: Reconfiguration plan to introduce an activity for load balancing

```
1  int RID = BeginReConfiguration("Fa", "Fb", "Fb1");
2     BeginAwaReConfig(RID, "Fa");
3        AddOutputLink(RID, "Fa", "ao1", new String[]{"b1i1"});
4        ChangeOutputStrategy(RID, "Fa", "ao1", "RoundRobin");
5     int it1 = EndAwaReConfig(RID, "Fa");
6     BeginAwaReConfig(RID, "Fb");
7        ChangeInputOrder(RID, "Fb", "bi1", "Sequence");
8     int it2 = EndAwaReConfig(RID, "Fb");
9     LaunchActivity("Fb1Spec.xml", "Fb1");
10    BeginAwaReConfig(RID, "Fb1");
11       StartExec(RID, "Fb1");
12       ChangeInputOrder(RID, "Fb1", "b1i1", "Sequence");
13    int it3 = EndAwaReConfig(RID, "Fb1");
14    int[] AgreementSet=new int[]{it1, it2, it3};
15 int K=EndReConfiguration(RID, AgreementSet);
```

To improve the effects of the load balancing, later and as many times as necessary it is possible to increase the number of replicas of the *Fb* activity.

In Figure 4.21 a scenario is depicted where one more activity, named *Fb*2, is introduced as a new replica of the *Fb* activity.

The possibility of introducing multiple and variable number of activities as replicas for load balancing purposes hinders the determinism of knowing the maximum number of iterations for each replica activity.

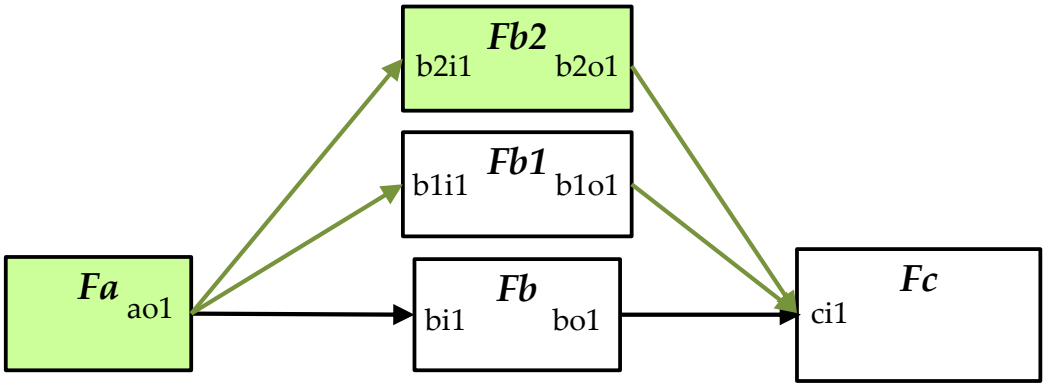


Figure 4.21: Introduce one more activity to increase the load balancing effects

The script of the reconfiguration plan for introducing one more activity, as depicted in Figure 4.21, is presented in Listing 4.13.

Note that the output strategy of the *ao*1 output port in the *Fa* activity keeps unchanged as *RoundRobin*, as previously defined in Listing 4.12.

Listing 4.13: Reconfiguration plan to introduce the *Fb*2 activity as one more replica

```
int RID = BeginReConfiguration("Fa", "Fb2");
    BeginAwaReConfig(RID, "Fa");
       AddOutputLink(RID, "Fa", "ao1", new String[]{"b2i1"});
    int it1 = EndAwaReConfig(RID, "Fa");
    LaunchActivity("Fb2Spec.xml", "Fb2");
    BeginAwaReConfig(RID, "Fb2");
       StartExec(RID, "Fb2");
       ChangeInputOrder(RID, "Fb2", "b2i1", "Sequence");
    int it2 = EndAwaReConfig(RID, "Fb2");
    int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID, AgreementSet);
```

4.4.6 Scenario 6: Recovering from Faults

Long-running scientific experiments involving data streaming can be supported by long-running workflows characterized by multiple activities for processing data sets. These activities execute multiple, eventually infinite number of iterations and their *Tasks* can access external resources often unreliably or with limitations on the quality of service that can cause faults. After a fault has occurred, the most common approach followed by some widely used workflow systems, for instance Kepler [Kep14], requires restarting the

entire workflow anew, which can lead to a waste of execution time due to unnecessarily repetition of computations. However, by using dynamic reconfigurations of long-running AWARD workflows some activity faults can be recovered in order to avoid restarting the complete workflow. For illustrating fault recovery scenarios let us consider the workflow presented in Figure 4.22, where during the execution of the multiple iterations any of the three activities can fail.

The *Fa* activity can fail when it puts tokens into the AWARD Space (*output* state of the *State Machine*), the *Fb* activity can fail during *Task* invocation (*invoke* state of the *State Machine*) and the *Fc* activity can fail when it gets tokens from the AWARD Space (*input* state of the *State Machine*).

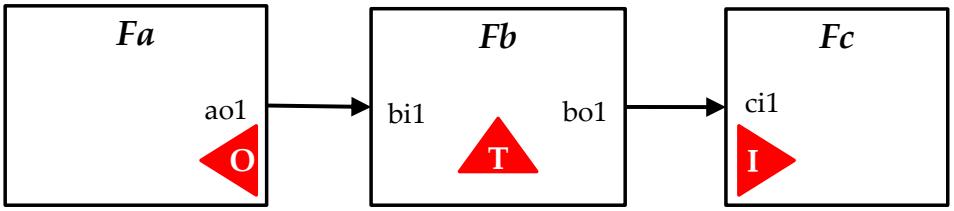


Figure 4.22: Distinct failures (Output, Task and Input) on workflow activities

Each activity can fail separately on distinct iterations. Therefore it may not be possible to apply a reconfiguration plan involving the three activities.

However, according to the analysis performed by the workflow developer three possible reconfiguration plans can be applied for recovering faults respectively for the *Fa*, *Fb* and *Fc* activities, as presented respectively in Listing 4.14, Listing 4.15 and Listing 4.16.

Listing 4.14: Recover a fault when *Fa* is in the *output* state

```
int RID = BeginReConfiguration("Fa");
    BeginAwaReConfig(RID, "Fa");
         RetryAfterFaultOut(RID, "Fa");
    int it1 = EndAwaReConfig(RID, "Fa");
    int[] AgreementSet=new int[]{it1};
int K=EndReConfiguration(RID, AgreementSet);
```

Listing 4.15: Recover a *Task* fault when *Fb* is in the *invoke* state

```
int RID = BeginReConfiguration("Fb");
    BeginAwaReConfig(RID, "Fb");
         ChangeTask(RID, "Fb", "scenariotasks.TaskFbnew");
         RetryAfterFaultTask(RID, "Fb");
    int it1 = EndAwaReConfig(RID, "Fb");
    int[] AgreementSet=new int[]{it1};
int K=EndReConfiguration(RID, AgreementSet);
```

Listing 4.16: Recover a fault when *Fc* is in the *input* state

```
int RID = BeginReConfiguration("Fc");
    BeginAwaReConfig(RID, "Fc");
         RetryAfterFaultIn(RID, "Fc");
    int it1 = EndAwaReConfig(RID, "Fc");
    int[] AgreementSet=new int[]{it1};
int K=EndReConfiguration(RID, AgreementSet);
```

The plan in Listing 4.14 for recovering the *Fa* activity assumes that any token had not yet been written in the *output* state. This requires that failure problems related to the AWARD Space access are solved and possible clean up actions performed on the AWARD Space.

The plan in Listing 4.15 for recovering the *Fb* activity assumes that the workflow developer specifies a new activity *Task* for solving the problem that caused the failure.

The plan in Listing 4.16 for recovering the *Fc* activity assumes that any token had not yet been read in the input state. This requires that problems related to the AWARD Space access are solved and possible restore actions performed on the AWARD Space.

4.4.7 A Note on the Soundness of the Reconfiguration Scenarios

The workflow reconfigurations reached from applying the scenarios 1 to 4 presented in the previous sections satisfy the soundness properties (Definition 4.4 on page 94). However, in scenario 5 (Section 4.4.5) the 1st soundness property (*Implicit termination*) is not guaranteed for activities that are processing tokens in a load balancing scenario. In fact, in workflow of scenario 5, the *Fa* activity issues tokens in round-robin fashion to be received by the downstream *Fb*, *Fb*1 and *Fb*2 activities that are running separately without any knowledge about the number of tokens sent by the *Fa* activity to each of the downstream *Fb*, *Fb*1 and *Fb*2 activities. Therefore when the *Fa* activity terminates by reaching its maximum number of iterations, the downstream *Fb*, *Fb*1 and *Fb*2 load balancing activities cannot terminate because they are waiting for the next sequence token. In this scenario all load balancing activities must be terminated separately by using an explicit asynchronous forced termination command. In scenario 6 the satisfaction of soundness properties is dependent on the success of the reconfiguration plan for recovering the activity faults. For the case of *Task* invocation fault if the new *Task* corrects the problem that originated the fault the reconfiguration is completely compliant with the soundness properties. For the cases of input or output faults if the actions for restoring the AWARD Space are unsuccessful the faults may become persistent requiring to explicitly force the termination of the workflow activities, which violates the 1st soundness property (*Implicit termination*).

4.5 Chapter Conclusions

The support for dynamic reconfigurations is an innovative contribution of the AWARD model. It is strongly justified by real scientific applications involving long-running workflows that after a long elapsed execution time may require dynamic changes for adjusting the structure or the behavior of the workflow. In many scenarios of real scientific applications the requirements are not well known at workflow design time so it is important to be able to change the workflow without the need of discarding the results of the previous workflow processing.

The AWARD model and a corresponding extension to the AWARD machine support the concept of dynamic reconfiguration plan for submitting sequences of reconfiguration operators to allow structural and behavioral workflow changes. When a reconfiguration plan involves multiple workflow activities a two phase commit protocol is used for achieving the earliest iteration number where all activities apply their part of the reconfiguration plan.

This chapter presented the extensions to the AWARD machine for supporting a set of dynamic reconfiguration operators allowing structural and behavioral changes of the workflow. The semantics of each operator as well as the semantics of dynamic reconfiguration plans are also presented. The chapter terminates with a discussion of a set of useful workflow scenarios illustrating the flexibility of the AWARD model for supporting dynamic reconfigurations.

All of the presented scenarios as well as all application examples presented later in Chapter 6 were actually implemented and executed, supported by the AWARD architecture, whose design and implementation is discussed in detail in the following chapter.

Chapter 5

The Architecture and Implementation of the AWARD Machine

The architecture and implementation of the AWARD framework, which allows the development and execution of scientific workflows based on the AWARD model.

An important contribution of this dissertation is providing the implementation of an operational framework for allowing end users to be able to develop and execute their workflows without the need to know about low-level details related to the AWARD machine.

This chapter describes the AWARD framework for supporting the development and the execution of real scientific workflows based on the AWARD model. The software architecture and the corresponding implementation of the AWARD framework components are based on object-oriented software design and rely on XML and Java technologies.

In Chapter 6 we describe how the implementation of the AWARD framework and a set of associated tools have been used for developing workflows in distinct application cases, namely a text mining application, which was entirely developed by an external user, relying upon AWARD to experiment with alternative solutions.

According to the standards, in particular the IEEE 1471-2000 [IEE12], a software architecture of a system must describe the principles governing its design as well as the fundamental organization of the system in terms of its components, their relationships to each other and the dependencies upon the underlying execution environment.

Therefore in Section 5.1 we discuss the rationale, principles and assumptions for developing the AWARD framework to support running AWARD workflows on distinct

and heterogeneous execution platforms, from standalone computers to processing nodes on cluster or cloud infrastructures.

In Section 5.2 we describe the AWARD framework. It is organized with a top layer, the *Abstract Workflows* layer, for defining the concerns involved when developers are designing the AWARD workflows and a bottom layer, the *Concrete Workflows* layer, including the operational components, namely the AWARD Space and the AWARD machine as an *Autonomic Controller* for allowing the execution of workflows on computational resources.

The life-cycle for developing AWARD workflows is described in Section 5.3.

In Section 5.4 we describe the grammar for specifying AWARD workflows using the XML Schema Definition (XSD) language [W3C15].

In Section 5.5 we discuss the requirements for implementing the AWARD Space and the reasons why we have chosen an implementation based on the IBM TSpaces [Leh+01], which is itself based on the Linda model [CG89].

In Section 5.6 we describe the *DynamicLibrary.jar* and its application interface (*DynamicAPI*) for providing the set of reconfiguration operators that must be integrated in any Java application for submitting dynamic reconfiguration plans.

Section 5.7 outlines the main software components of the AWARD machine, and the structure of the XML file used for supporting the configuration of the execution environment, the main object classes and their interactions to implement the architecture of the AWARD machine as an AWA executor, including the integration of a *Rules Engine*.

The set of handlers supported by the *Autonomic Controller* for receiving asynchronous notifications, such as for requesting the *AWA Context*, for forcing an AWA termination and for supporting dynamic reconfigurations are described in Section 5.8.

Section 5.9 describes the set of tools designed and implemented for running and monitoring AWARD workflows.

The chapter conclusions are presented in Section 5.10.

5.1 Assumptions for Implementing the AWARD Framework

Unfortunately many scientific workflow systems and tools lack the needed flexibility to install, configure and execute workflows on heterogeneous infrastructures. In most cases there are dependencies on software components, middleware platforms or database systems that preclude easy replication of these workflow systems on multiple and heterogeneous computing nodes.

Therefore from the beginning of our research we intended to provide an operational implementation of the AWARD model as a working prototype with minimal dependencies on disparate technologies and decoupled from any particular execution infrastructure. Then we settled the following set of requirements and assumptions to implement the AWARD framework:

✧ **Implementation requirements**

- A workflow activity (AWA) must be executed in distinct infrastructures with heterogeneous operating systems and decoupled from disparate technologies for allowing the workflow execution in a diversity of infrastructures, ranging from a standalone Windows or Linux-based computer system to computing nodes in clusters and clouds. Therefore Java technologies were chosen;
- All configuration data and the AWARD workflow specification must be described using standard markup languages in order to facilitate their edition, parsing validation and memory loading. Therefore the XML language and the corresponding XML schema were chosen;
- The executable program which encapsulates the *Autonomic Controller* of an AWA activity should be as transparent as possible regarding the networking and the related addressing issues;
- Activity *Tasks* development for specific problem domains should be able to use any type of Java software libraries independently of the AWARD machine. Then the specification of AWARD workflows may include the file system location of these libraries to be loaded under the control of the *Autonomic Controller* when it executes an activity *Task*;
- The end user tools for launching and monitoring the workflow execution must be available as flexible commands able to be submitted to local computers or to remote computing nodes. As an example the command to launch the workflow AWA activities must be able to launch one AWA activity or groups of AWA activities in distinct computing nodes.

5.2 The AWARD Framework

The AWARD framework (presented in Figure 5.1) encompasses the concepts and mechanisms for the development and execution of AWARD workflows and is subdivided in two main levels:

1. The *Abstract Workflows* level defines the high-level abstractions, such as what is an AWARD workflow, links as abstractions over the AWARD Space for connecting the workflow activities and a set of dynamic reconfiguration operators for supporting workflow reconfiguration plans;
2. The *Concrete Workflows* level offers an operational implementation of the AWARD model and a set of software tools for facilitating the life-cycle of AWARD workflows

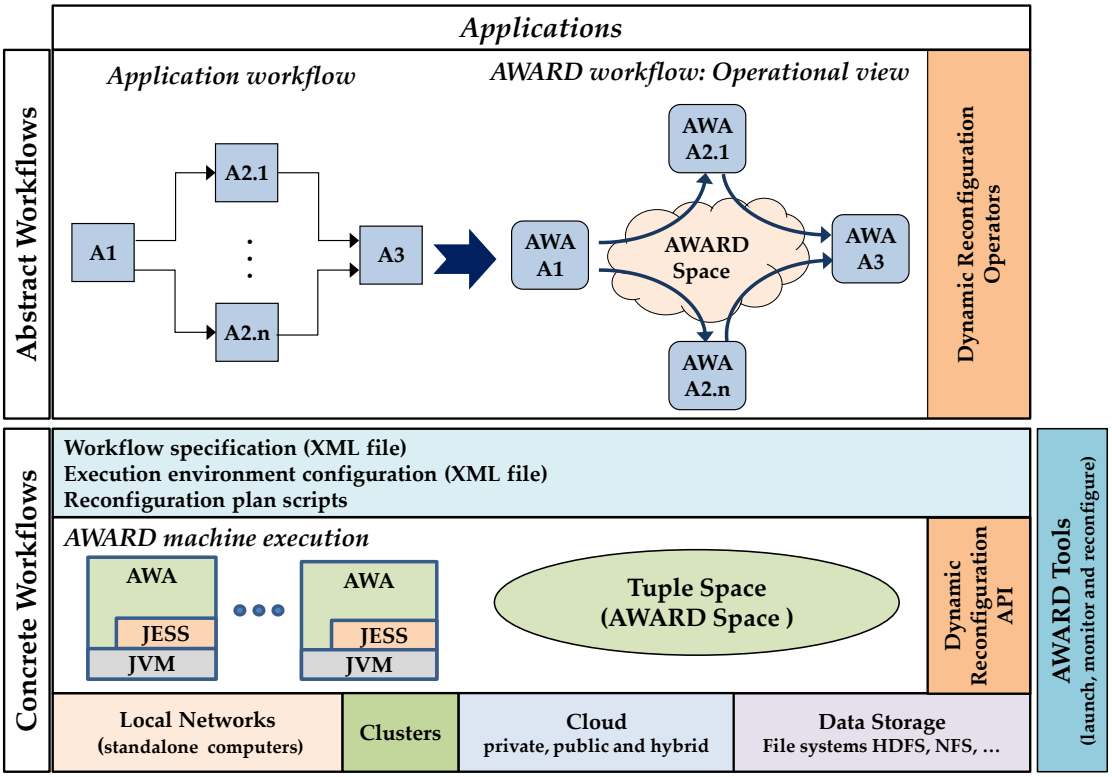


Figure 5.1: The AWARD framework

development. The upper side of this level defines how to specify concrete workflows using a specific XML schema as well as a concrete configuration for the execution environment. This includes, for instance, the configuration of the access to the AWARD Space and the underlying infrastructure, namely the available computing nodes, and the configuration of the software libraries possibly used in the *Task* development and required to execute the workflows. This level provides the core components for executing concrete workflows:

- The implementation of the AWARD machine as an *Autonomic Controller* capable of executing any workflow activity, including the support for handling the activity dynamic reconfiguration;
- The implementation of the AWARD Space as a tuple space following the Linda model [CG89];
- The dynamic reconfiguration interface (*DynamicAPI*), which encapsulates the dynamic reconfiguration operators allowing the development of scripts for applying reconfiguration plans;
- The AWARD tools, available as a set of commands, for supporting the end users in launching and monitoring the execution of workflow activities on the underlying infrastructures such as local network standalone computers, clusters, clouds and allowing interfacing with distinct data storage mechanisms such as

local file systems, distributed file systems such as the Hadoop Distributed File System (HDFS) [Apa15a] or the Network File System (NFS) [San+88] or other middleware for managing distributed data repositories.

5.3 The Life-cycle for Developing AWARD Workflows

An important motivation that led to the creation of the AWARD model aimed at simplifying the life-cycle of scientific workflows by allowing the workflow developers to focus on the application requirements and not on details of the workflow system implementation.

In general the life-cycle of scientific workflows includes the application decomposition into activities and the design of each activity by the selection of adequate programs or software components for executing these activities, as well as the configuration setup of the connections for supporting the data or control flows between activities. After that stage the workflow instances are launched for execution on the underlying infrastructures. Usually there is a lack of support for monitoring the activities execution until their termination. Therefore the AWARD workflow life-cycle is based on eight steps as depicted in Figure 5.2 where the 7^{th} step is used to provide an adequate support for monitoring the workflow execution and, as a distinct characteristic of the AWARD model, the 8^{th} step supports the development and submission of reconfigurations plans for allowing dynamic workflow reconfigurations.

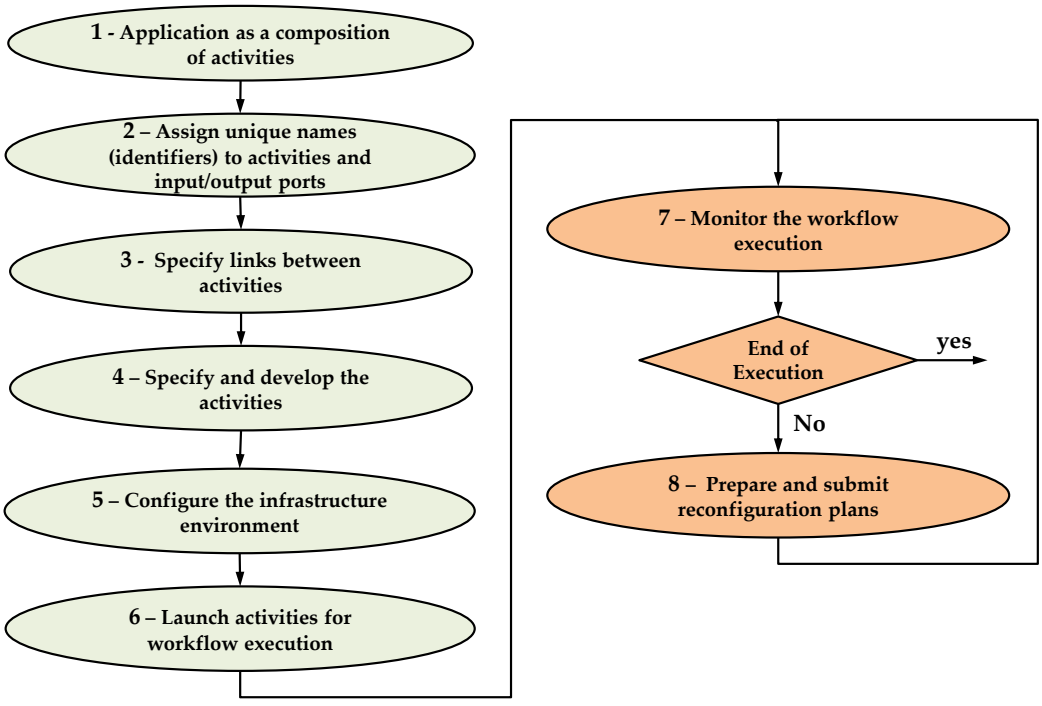


Figure 5.2: The life-cycle of the AWARD workflows

The main actions, performed by workflow developers on each step of the life-cycle, are described in the following:

Step 1: According to the problem domain requirements the application is decomposed into multiple activities and their dependencies are specified in terms of sequential or parallel patterns. The activity composition includes the links for connecting the output ports of an activity to the input ports of other activities. The result of this step is a graph design, the AWARD workflow, where the graph vertices are the workflow activities and the graph edges are the links between activities. The AWARD workflow has a logical name related to the application and a global maximum number of iterations for all activities;

Step 2: According to the AWARD model this step corresponds to assigning names as unique identifiers to each activity, to each input port and to each output port. Currently the AWARD architecture assumes that the uniqueness of names is ensured at design time by the workflow developer;

Step 3: For each link the associated token type is defined by specifying an application dependent data type used to exchange data or control flow between activities. In the case of a simple link, the output port (the origin of link) is associated with the name of the destination input port and the token mode is specified as *Single*. In addition to specifying the set of associated destination input ports, the output port (the origin of link) of a multi-link also specifies the token mode as *Replicate* or *RoundRobin*. The destination input port of a simple link also specifies the input mode as *Iteration* or *Sequence*. If the input mode is *Sequence*, then this restricts the corresponding activity to a single input port, which can only be connected through a simple link. In the case of a multi-link the input mode of a destination input port is *Iteration* or *Any*;

Step 4: For each activity its internal components are specified and developed according to the substeps presented in Figure 5.3.

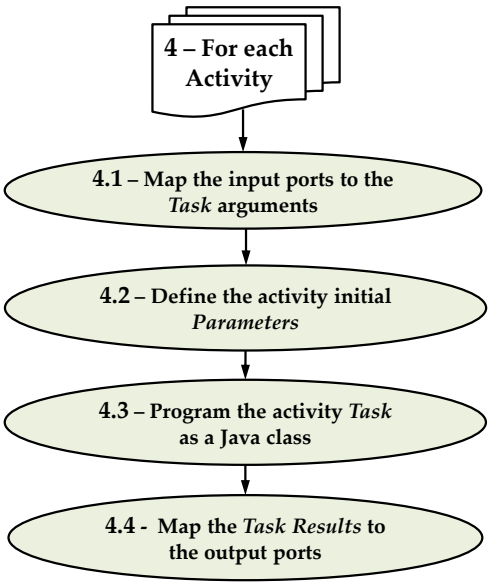


Figure 5.3: Substeps for specifying and developing the activities

Step 4.1: An activity *Task* can receive an array of *Nargs* arguments. A map function defines how tokens received on the activity input ports are mapped to *Task Arguments* by a set of pairs where a pair associates the argument number *0..Nargs* to the name of an input port;

Step 4.2: An activity can receive an array of *Npars* initial *Parameters*, for instance filenames, URLs, IP addresses, IP ports etc;

Step 4.3: Each activity resulting from the application decomposition has an associated *Task*, which represents the executable code of the activity algorithm. The workflow developer specifies, in the Java language, the entry point of the activity *Task* class according to the skeleton presented in Listing 5.1;

Step 4.4: An activity *Task* can return an array of *Nres Results*. A map function defines how these *Results* are mapped to the output ports by a set of pairs where a pair associates the name of an output port to an result number *0..Nres*.

Listing 5.1: The entry point class for implementing an activity *Task*

```
1  // Entry point class for implementing an activity Task
2  public class ActivityTask implements IGenericTask {
3
4      public Object[] EntryPoint(Object[] Arguments, Object[] Parameters) {
5          // Any Java code, including the invocation of the class methods
6          // for modeling the activity algorithm, the utilization
7          // of application-dependent software libraries and other resources,
8          // for instance the possible access to Web services
9
10         Object[] Results = new Object[]{ res0,...,resN };
11         return Results;
12     }
13     // Other class methods for modeling the activity algorithm
14 }
```

Step 5: The AWARD workflows can be launched in distinct infrastructures, including a single computer, multiple computers in a local network or multiple computing nodes as virtual machines, for instance in clusters or clouds. Therefore before launching a workflow some configuration details of the underlying infrastructure must be defined, for instance, the addressing details of the AWARD Space, the working directories for accessing data and executable files or the home directories for software libraries needed for running some activities. An important characteristic of the configuration environment is related to the levels of logging for debugging or monitoring the workflow execution. These configurations are specified in a configuration file named *awardConfig.xml* whose schema is presented in Section 5.7.1 (Listing 5.10 on page 168);

Step 6: The workflow is launched for execution on the available underlying infrastructure. The AWARD framework offers flexible tools to launch all activities in the same computer or groups of activities including one by one on separate computing nodes, for instance virtual machines in clusters or clouds;

Step 7: The AWARD configuration environment allows specifying different levels of logging for enriching the information available during the workflow execution. Therefore in this step the workflow developer can perform the monitoring of the workflow execution with great flexibility;

Step 8: As presented in Chapter 4 the AWARD model supports dynamic workflow reconfigurations. Following the analysis of logging information performed in the 7^{th} step by monitoring the workflow execution or by analyzing the workflow intermediate results, the developer can prepare reconfiguration plans in order to change the structure or the behavior of the workflow. Therefore the 7^{th} and 8^{th} steps can be repeated cyclically until the termination of the workflow execution.

5.4 The Workflow Specification XML Schema

The flexibility and usability of user interfaces to design scientific workflows are important issues, namely when the workflows have a large number of activities. Other important issue is the lack of standards related to workflow specification languages. Initiatives like extensions to the Business Process Execution Language (BPEL) [BPE06], the Yet Another Workflow Language (YAWL) [RH09] as a workflow language inspired in Petri nets, the UML (Unified Modeling Language) [OMG15] diagrams or in most cases some specific XML schemas [W3C15] have been used without consensus. These issues are out of the scope of this dissertation.

However, in our work we found out that the flexibility of the standard XML Schema Definition (XSD) is sufficient for defining markup languages for specifying the workflow activities and their interactions as well as other workflow configurations.

Specifying an AWARD workflow consists of editing a XML file containing the definition of values for workflow details where the structure of the XML file is conforming to a XML schema. This schema defines the grammar for workflow specification and allows validating if a given workflow XML file is or is not well-formed. In order to facilitate the description in the following the AWARD XML schema is presented by blocks using a graphical representation and the corresponding syntax in XSD (XML Schema Definition), as recommended by the *World Wide Web Consortium* (W3C) for specifying how to formally describe the elements in an *Extensible Markup Language* (XML) document.

As simple but sufficient view of the XSD syntax there are few main concepts: i) The XML *namespaces* as definitions for the scope of names; and ii) The *ComplexType* to define an entity type as a *sequence* of named *element* of primitive types (*int*, *string*, etc.) or other *ComplexTypes*.

As depicted in Figure 5.4 and detailed in Listing 5.2 the specification of an AWARD workflow (element named *AwardWorkflow* in line 7 of the listing) consists of assigning values to the elements of the *AwardSpecification complexType*. These elements are defined in the scope of AWARD specification XML *namespace*, denoted by *http://Award.xsd*.

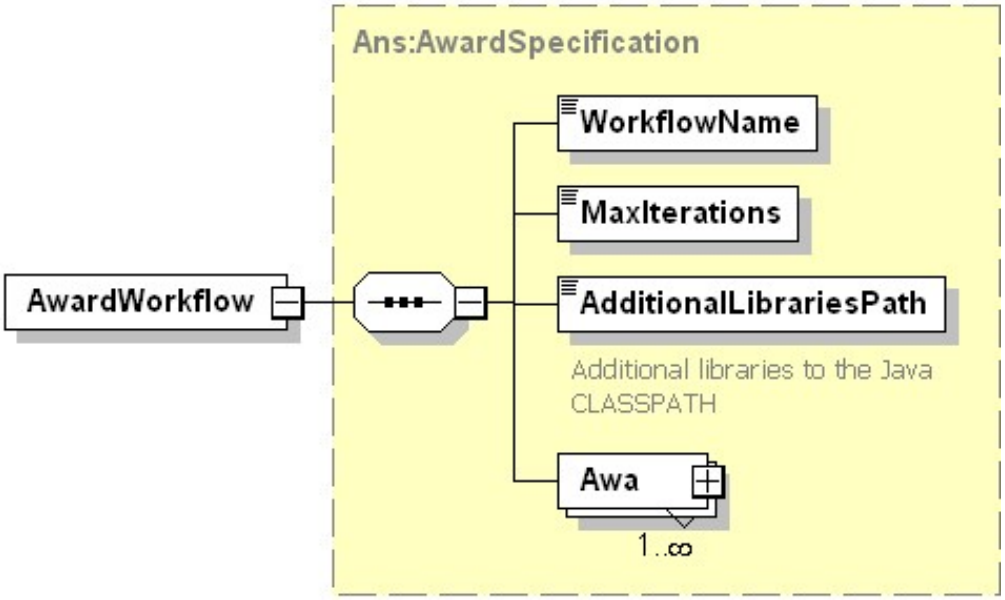


Figure 5.4: AWARD XML schema: The workflow specification

The first element named *WorkflowName* defines a name for the workflow as a *string*. The element named *MaxIterations* defines the maximum number of iterations as an integer number. The element named *AdditionalLibrariesPath* is a string containing a list of application dependent additional Java packages used to develop the *Tasks* of the workflow activities to be added to the Java CLASSPATH environment variable. The repeatable

Listing 5.2: AWARD XSD: The workflow specification

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
    targetNamespace="http://Award.xsd"
    elementFormDefault="unqualified"
    xmlns:Ans="http://Award.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" >
  <xs:element name="AwardWorkflow" type="Ans:AwardSpecification"/>
  <xs:complexType name="AwardSpecification">
    <xs:sequence>
      <xs:element name="WorkflowName" type="xs:string"/>
      <xs:element name="MaxIterations" type="xs:int"/>
      <xs:element name="AdditionalLibrariesPath" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Additional libraries to the Java CLASSPATH
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Awa" type="Ans:AwaSpecification"
                minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  . . .
</xs:schema>
```

element named *Awa* defines the specification of each workflow activity and is based on the *AwaSpecification complexType*.

The specification of an *Autonomic Workflow Activity* (AWA) (*AwaSpecification*), as depicted in Figure 5.5 and detailed in Listing 5.3, consists of assigning values to the elements of the A*waSpecification complexType*. The element named *name* defines a name for the AWA activity as a string. The element named *ControlUnit* is a *complexType* used to specify the initial configuration of the *State Machine*. The element named *input* with zero or more occurrences allows the specification of the AWA activity input ports where each input port is an instance of the *input complexType*. The element named *output* with zero or more occurrences allows the specification of the AWA activity output ports where each output port is an instance of the *output complexType*. The element named *Task* is a *complexType* used to specify the activity *Task*.

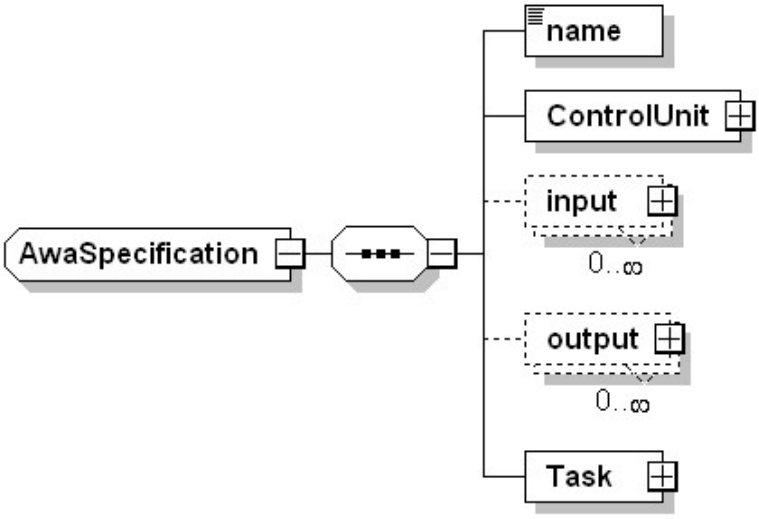


Figure 5.5: AWARD XML schema: The AWA specification

Listing 5.3: AWARD XSD: The AWA specification

```
<xs:complexType name="AwaSpecification">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="ControlUnit" type="Ans:ControlUnit"/>
    <xs:element name="input" type="Ans:InputPort" minOccurs="0"
                                                  maxOccurs="unbounded"/>
    <xs:element name="output" type="Ans:OutputPort"
              minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Task" type="Ans:Task" />
  </xs:sequence>
</xs:complexType>
```

As depicted in Figure 5.6 and detailed in Listing 5.4, the specification of the initial configuration for the *State Machine* (element named *ControlUnit*) consists of assigning

an optional particular value to the AWA *MaxIterations* that overrides the global value of *MaxIterations* defined for all workflow activities. The element named *InitialState* allows specifying (as a *string*) if the *State Machine* initialization is in *idle* state or in the *WaitConfig* state. The element named *RulesFilePath* is a string with the *pathname* of the text file containing the initial default facts and rules used by *Rules Engine* of the *Autonomic Controller*.

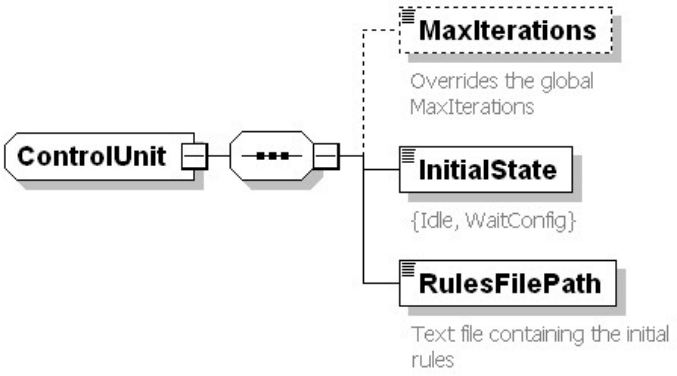


Figure 5.6: AWARD XML schema: The initial State Machine specification

Listing 5.4: AWARD XSD: The initial State Machine specification

```
<xs:complexType name="ControlUnit">
  <xs:sequence>
    <xs:element name="MaxIterations" type="xs:int" minOccurs="0" maxOccurs="1">
      <xs:annotation>
           <xs:documentation>
               Overrides the global MaxIterations
           </xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="InitialState" type="xs:string">
      <xs:annotation>
           <xs:documentation>{idle,WaitConfig}</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="RulesFilePath" type="xs:string">
       <xs:annotation>
          <xs:documentation>
            Text file containing the initial rules
          </xs:documentation>
       </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

As depicted in Figure 5.7 and detailed in Listing 5.5, the specification of an input port (*InputPort*) consists of assigning values to the elements of a *complexType* named *InputPort*. The element named *name* allows assigning a name to the input port as a

string. The element named *state* allows assigning one of the three possibly string values (*Enable*, *Disable*, *EnableFeedback*) to the initial state of the input port. The element named *tokenType* allows specifying the qualified name of the Java type used for tokens associated with the input port. The element named *orderMode* allows assigning one of the three possibly string values (*Iteration*, *Sequence*, *Any*) to specify the mode how the input port consumes tokens.

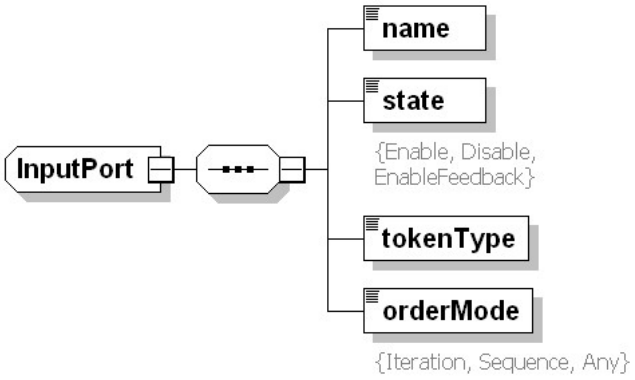


Figure 5.7: AWARD XML schema: The input port specification

Listing 5.5: AWARD XSD: The input port specification

```
<xs:complexType name="InputPort">
   <xs:sequence>
     <xs:element name="name" type="xs:string" />
     <xs:element name="state" type="xs:string" >
        <xs:annotation>
          <xs:documentation>{Enable, Disable, EnableFeedback}</xs:documentation>
         </xs:annotation>
     </xs:element>
     <xs:element name="tokenType" type="xs:string" />
     <xs:element name="orderMode" type="xs:string" >
        <xs:annotation>
          <xs:documentation>{Iteration, Sequence, Any}</xs:documentation>
        </xs:annotation>
     </xs:element>
   </xs:sequence>
 </xs:complexType>
```

As depicted in Figure 5.8 and detailed in Listing 5.6, the specification of an output port (*OutputPort*) consists of assigning values to the elements of a *complexType* named *OutputPort*. The element named *name* allows assigning a name to the output port as a string. The element named *state* allows assigning one of the three possibly string values (*Enable*, *Disable*, *EnableFeedback*) to the initial state of the output port. The element named *tokenType* allows to specify the qualified name of the Java type used for tokens associated with the output port. The element named *modeToken* allows assigning one of the three possibly string values (*Single*, *Replicate*, *RoundRobin*) to specify the mode how the output

port emits tokens. The element named *sendTo* allows specifying (as strings) one or more names of destination input ports connected to the output port.

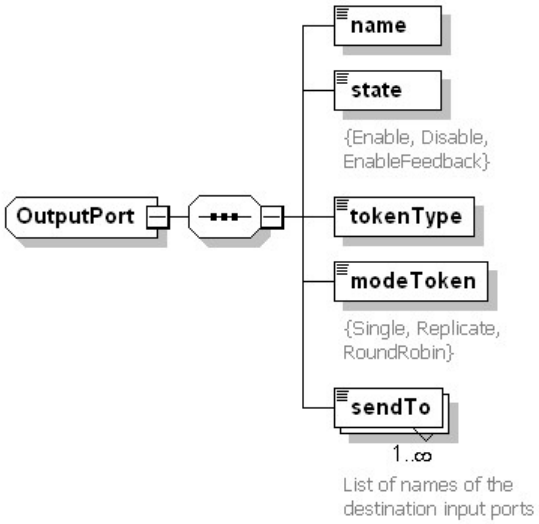


Figure 5.8: AWARD XML schema: The output port specification

Listing 5.6: AWARD XSD: The output port specification

```
<xs:complexType name="OutputPort">
   <xs:sequence>
     <xs:element name="name" type="xs:string" />
     <xs:element name="state" type="xs:string">
       <xs:annotation>
         <xs:documentation>{Enable, Disable, EnableFeedback}</xs:documentation>
       </xs:annotation>
     </xs:element>
     <xs:element name="tokenType" type="xs:string" />
     <xs:element name="modeToken" type="xs:string">
        <xs:annotation>
          <xs:documentation>{Single, Replicate, RoundRobin}</xs:documentation>
        </xs:annotation>
     </xs:element>
     <xs:element name="sendTo" type="xs:string" minOccurs="1"
                                              maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>
               List of names of the destination input ports
          </xs:documentation>
        </xs:annotation>
     </xs:element>
   </xs:sequence>
</xs:complexType>
```

As depicted in Figure 5.9 and detailed in Listing 5.7, the specification of the activity task (*Task*) consists of assigning values to the elements of a *complexType* named *Task*. The

optional element named *Parameters* allows defining one or more initial *Task* parameter where each parameter is a string named *par*. The element named *SoftwareComponent* allows specifying the Java software component implementing the activity algorithm as well as the mappings from input ports to *Task Arguments* and mappings from *Task Results* to the output ports.

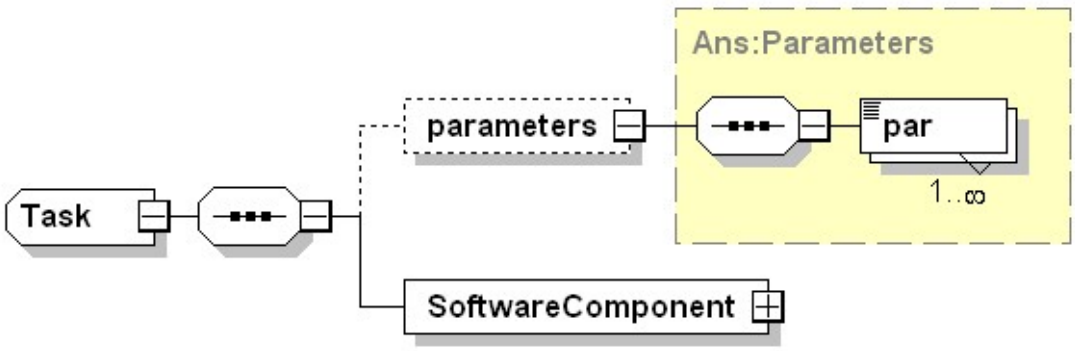


Figure 5.9: AWARD XML schema: The activity *Task* specification

Listing 5.7: AWARD XSD: The activity *Task* specification

```
<xs:complexType name="Task">
   <xs:sequence>
     <xs:element name="parameters" type="Ans:Parameters"
                                   minOccurs="0" maxOccurs="1" />
     <xs:element name="SoftwareComponent" type="Ans:SoftwareComponent" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="Parameters">
   <xs:sequence>
      <xs:element name="par" type="xs:string" minOccurs="1"
                                          maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>
```

As depicted in Figure 5.10 and detailed in Listing 5.8, the specification of the activity *Task* software component (*SoftwareComponent*) consists of assigning values to the elements of a *complexType* named *SoftwareComponent*. The element named *mappingArgs* allows specifying how *Task Arguments* are mapped from input ports as an unlimited sequence of elements named *arg*. Each element *arg* allows assigning values to the elements of a *complexType*named *arg* that represents an association between an input port named *inName* and the index position, named *idxArg*, on *Task Arguments* list.

The element named *taskImplementationType* allows to specify the qualified name of the Java type which implements the *Task* algorithm.

The element named *mappingResults* allows specifying how a list of *Task Results* are mapped to output ports as an unlimited sequence of elements named *result*. Each element

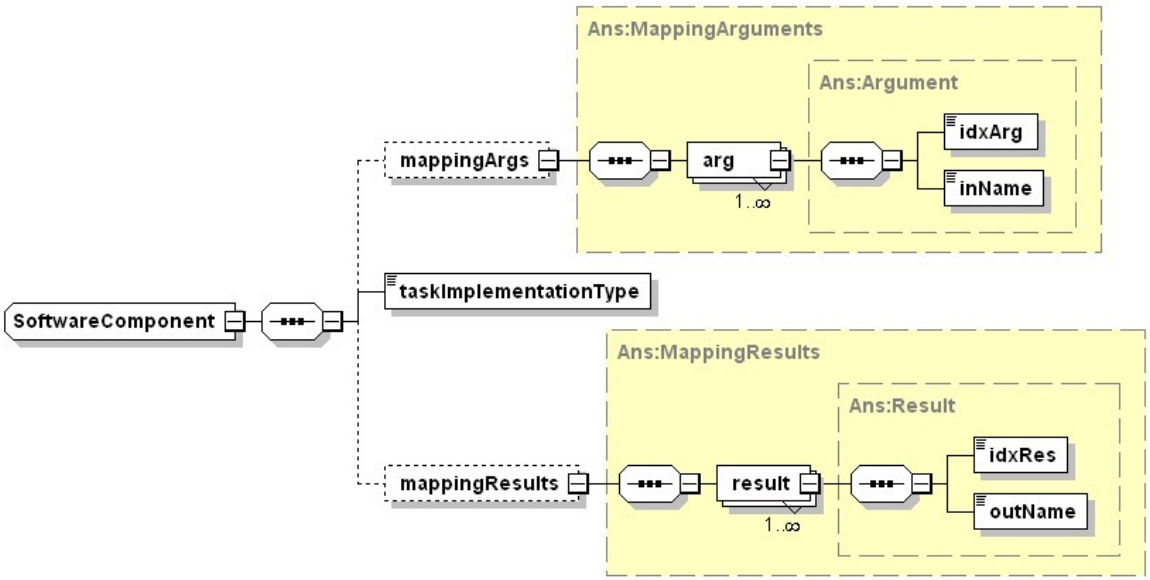


Figure 5.10: AWARD XML schema: The *Task* software component specification

Listing 5.8: AWARD XSD: The *Task* software component specification

```
<xs:complexType name="SoftwareComponent">
  <xs:sequence>
    <xs:element name="mappingArgs" type="Ans:MappingArguments"
                                          minOccurs="0" maxOccurs="1"/>
      <xs:element name="taskImplementationType" type="xs:string"/>
      <xs:element name="mappingResults" type="Ans:MappingResults"
                                          minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MappingArguments">
  <xs:sequence>
    <xs:element name="arg" type="Ans:Argument"
                                    minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="Argument">
  <xs:sequence>
     <xs:element name="idxArg" type="xs:int"/>
     <xs:element name="inName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MappingResults">
  <xs:sequence>
    <xs:element name="result" type="Ans:Result"
                                    minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="Result">
    <xs:sequence>
      <xs:element name="idxRes" type="xs:int"/>
      <xs:element name="outName" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

result allows to assign values to the elements of a *complexType* named *result* that represents an association between a position index, named *idxRes*, of the $Task\ Results$ and an output port named *outName*.

For representing the data associated to the AWA specification details the implementation of the *Autonomic Controller* internally uses a set of objects as presented in Section 5.7 (Figure 5.15 on page 170).

✧ **Discussion**

Nowadays there are a lot of tools to edit XML files and validate their conformity to a XML schema. Therefore editing XML workflow files and their validation against the AWARD XML schema is out of the scope of this disertation. However, we developed a basic tool, included in the AWARD tools described in Section 5.9 for helping workflow developers to verify if a workflow specification is conform to the XML specification schema.

The development of a graphical tool for designing the AWARD workflows is also out of the scope of this dissertation.

5.5 The AWARD Space

In the AWARD model the AWARD Space abstraction supports the communication (links) and the corresponding tokens between input and output ports of the workflow activities.

Furthermore, the AWARD Space also supports the asynchronous event communication with workflow activities for forcing the termination of an AWA, for requesting the activity context information or for supporting the dynamic reconfiguration of activities.

The AWARD Space can be implemented as any globally accessed data store characterized by the properties defined in Section 3.4.2 on page 71 and outlined in the following:

- **Unbounded size:** Support for storing all tokens of all links independently of the token production pace by all workflow activities;
- **Associative (Content addressable):** The access to tokens must be associative for allowing token retrieval by any of their fields;
- **Atomic write:** When a producer writes a token into the AWARD Space, the consumers can not observe uncomplete sets of the token fields;
- **Persistent write:** The AWARD Space must ensure the persistence of tokens on non-volatile memory in a transparent way;
- **Atomic retrieval:** Token retrieval from the AWARD Space must be atomic in the presence of concurrency;
- **Subscribing:** The AWARD Space must allow implementing the observer software design pattern or the publish/subscribe messaging pattern, by maintaining a list of registered subscribers interested in specific tokens. When these tokens are written into the AWARD Space the subscribers are automatically and asynchronously notified. This property is mainly used to implement the event handlers for supporting the asynchronous communication to the *Autonomic Controller*, namely on dynamic reconfigurations.

For implementing the AWARD Space many distinct alternatives are available, for example multiple *Data Base Management Systems* (DBMS), multiple *Message Oriented Middleware* (MOM) based in message queues, or multiple in-memory key-value storage systems.

As examples (and considering only alternatives without license costs) we have several DBMS systems, such as MySQL [WA02], several MOM implementations based on the standard *Advanced Message Queuing Protocol* (AMQP) [OAS15a], for instance Apache Qpid [Apa15b] or based on *Java Message Service* (JMS) specification [Jav15] and multiple in-memory key value data storage, for instance Cassandra [LM10].

However, most of these possible approaches are closely coupled to specific infrastructures and are not flexible for installation in heterogeneous environments, for instance on

a standalone computer, on clusters or in computing nodes as virtual machines of distinct clouds.

On one hand the motivations for developing the AWARD model are the flexibility for decoupling the workflow execution from particular infrastructures and their configurations for instance the execution of distinct workflow activities in different computing nodes should not be dependent on the prior installation of technologies except for the Java runtime. On the other hand the properties required by the AWARD Space are closely related to the tuple space concept based on Linda model [CG89].

Then conceptually the AWARD Space can be implemented by any tuple based associative memory from several existing solutions. As examples, JavaSpaces [Ora12], IBM TSpaces [Leh+01], MozartSpaces [Spa12], Gigaspaces [Gig12], and SQLspaces [Col12] or even distributed tuple spaces, such as Comet [LP05] and Tupleware [Atk08], a distributed tuple space for cluster computing. However, most of those tuple space implementations are unavailable for reutilization or have dependencies upon other technologies. Although currently (as of 2016) the IBM TSpaces is unavailable, at the beginning of the first AWARD prototype implementation it was available and was chosen for the following reasons:

- The IBM TSpaces only depends on the Java language runtime;
- The IBM TSpaces software library is provided by a Java package (*tspaces.jar*) with a very small size of 411 KB, which enables that potentially any Java application can host a tuple space server or can access other remote tuple space servers;
- The communication for accessing the tuple space server is supported by Transmission Control Protocol/Internet Protocol (TCP/IP) connections, using a binary data format, which does not introduce great overheads;
- The IBM TSpaces server has the built-in functionality to support access via Hyper Text Transfer Protocol (HTTP) for accessing the status of the server. This is an important characteristic for monitoring the execution of long-running workflows by inspecting the tuples stored into the tuple space server containing the AWARD Space. Furthermore, due to the property of a unbounded size of the AWARD Space, acessing the status of the tuple space server allows to monitor the memory occupied by tokens, which is important to anticipate possibe problems related to physical memory limits, namely in workflows where some activities consume tokens at a slow pace when comparing with a high pace of the associated upstream activities

5.5.1 The AWARD Space as a Set of Tuple Spaces

The architecture of the AWARD Space implementation is presented in Figure 5.11. The AWARD Space is a Java application server that can be executed on any computing node and is accessible using two ports: a TCP/IP port used for supporting the operations performed by workflow activities (AWA) on a set of tuple spaces; and an HTTP port used

for supporting the server state monitoring including the visualization of tuples that are stored on all tuple spaces.

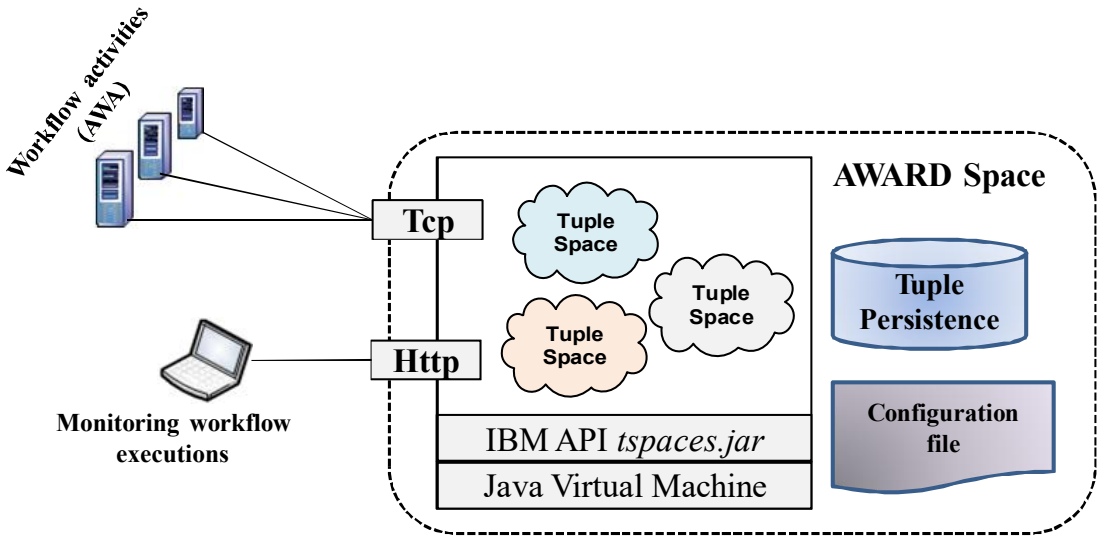


Figure 5.11: The AWARD Space implementation based on IBM TSpaces

The AWARD Space implementation uses distinct tuple spaces, such as a main tuple space for supporting the tokens flow between input and output ports and a set of tuple spaces for logging status information during the execution of the workflow activities as well as for logging information related to the elapsed execution time of each iteration of the workflow activities.

The IBM TSpaces supports the persistence of tuples in a relational database or in a file in the current server directory. In order to decouple the AWARD Space from database technologies the persistence of tuples is supported by a file store. This makes the installation of the AWARD Space server very simple only by copying files to any file system directory of any computing node only depending on the Java virtual machine.

After the copy of the required files and assuming that the destination directory is in the system PATH variable, launching an AWARD Space server is simply made by using the AWARD tool *"AwardSpace [tcpport=8500 httpport=8501]"* to submit a local or remote command to launch the server.

The definition of TCP/IP ports is optional to override the default ports values included in the configuration file as shown in Listing 5.9.

Despite the possibility for changing multiple parameters in the configuration of the IBM TSpaces server in the AWARD Space server we only change the accessing ports and disable the verification of permissions based on access control lists.

We developed a software library for accessing the AWARD Space to facilitate the development of the AWARD tools and any application, namely the workflow activities running in distinct computing nodes.

Listing 5.9: Excerpt of the AWARD Space server configuration file

```
1  [TCPServer]
2  TcpPort = 8200 // Tcp connection port
3  [HTTPServer]
4  HTTPServerSupport  = true // enable Http access on HttPort
5  HttpPort = 8201
6  [Users]
7  Sysadmin // user administrator
8  [AccessControl]
9  CheckPermissions = false // disable access control lists
10 AdminUser = sysadmin AdminPassword = admin
```

This library provides the following operations:

- *TSpace=CreateSpace(SpName, ServerName, TcpPort)*: It allows to initiate a tuple space named *SpName* on the AWARD Space running on a computer named *ServerName* and accessed on the *TcpPort* port. This allows to distinguish the spaces of different workflows or even different runs of the same workflow;
- *StoreTuple(TSpace, Tuple)*: A non-blocking operation which allows to write a *Tuple* into a previously created *TSpace*;
- *Tuple TakeTuple(TSpace, TupleTemplate)*: A blocking operation which allows to atomically taking (destructive read) a *Tuple* from the *TSpace*. The *TupleTemplate* tuple supports undefined fields denoted by (*?*) to define the tuple pattern matching. For instance, according to a token definition (Definition 3.4 on page 58) a tuple template can be interpreted as a token produced on the K^{th} iteration bound for the input port named I_1A with any token value and any sequence number;
- *RegisterEventHandler(TSpace, Handler, TupleTemplate)*: An operation to register a callback event handler (*Handler*) activated when any tuple that conforms to the tuple template *TupleTemplate* is written into the *TSpace*. This operation plays an important role for the support of dynamic interactions with the workflow activities, namely on dynamic reconfigurations.

5.6 Dynamic Reconfiguration API

As presented in section 3.4 the AWARD Space supports the links for connecting the input and output ports of the workflow activities. Additionally the AWARD Space has properties for allowing dynamic interactions based on the publishing/subscribing model.

The AWARD model is neutral regarding the way how these events are published into the AWARD Space which allows flexible approaches. The implemented approach allows any tool as a Java application to publish these events using a reusable Java library named *DynamicLibrary.jar*, which implements an application programming interface named *DynamicAPI*. This *Dynamic Reconfiguration API* (*DynamicAPI*) provides the functionality of

a set of dynamic reconfiguration operators to be integrated in development environments for developing tools, such as applications used to perform reconfiguration plans. For instance in Chapter 6 we describe a scenario that has also been presented in [AC13] where a tool monitors the execution of a workflow for detecting faults and automatically uses the *Dynamic Reconfiguration API*, to reconfigure the workflow in order to recover from faults.

For each reconfiguration operator the *Dynamic Reconfiguration API* encapsulates the interactions with the AWARD Space for allowing the AWA autonomic activities involved in reconfiguration plans to handle their reconfiguration sequence and achieve the global common iteration agreement needed to the success of a reconfiguration plan. These interactions are detailed in Section 5.8 on page 177 where the AWARD machine handlers are described.

5.6.1 Dynamic Reconfiguration Operators in the DynamicLibrary

The Java library *DynamicLibrary.jar* that implements the *DynamicAPI* interface must be used by the AWARD workflows developer when developing any kind of tools to monitor and reconfigure long-running workflows.

The library provides a set of functions corresponding to the dynamic operators defined in Section 4.2.1 and depicted in diagram of Figure 5.12.

Each function corresponding to a dynamic reconfiguration operator of the *DynamicAPI* interface, including the operator signature (arguments and respective types) is presented in Figure 5.13 on page 167.

Note that the $CreateOutput$ function has two versions with different signatures. This allows to create an output port without specifying the destination links ($SendTo$), that must be configured later using the $AddOutputLink$ or $ChangeOutputLink$ operators.

The description of arguments is presented in Table 5.1 on page 166.

Currently, the AWARD architecture and its implementation requires the developer who is responsible for the programming of the reconfiguration plan to ensure that the names of new activities and new input output ports are unique.

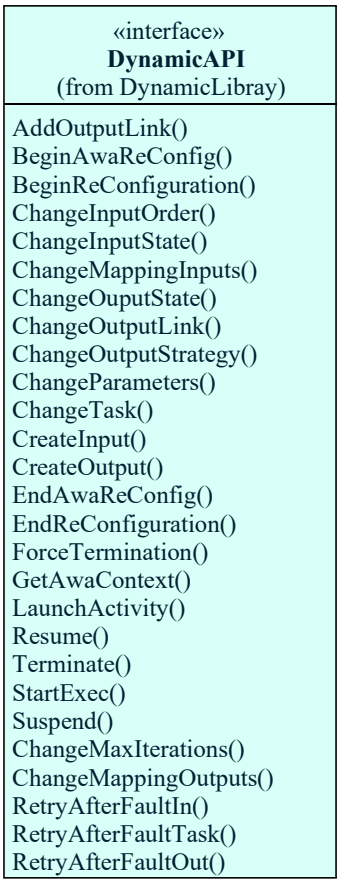


Figure 5.12: The interface provided by *DynamicLibray.jar*

Table 5.1: The arguments of the dynamic reconfiguration operators

Argument	**Description**
RID	Reconfiguration identifier
awaName	Activity name
inName	Input port name
outName	Output port name
dstList	List of input port names for adding to *sendTo* destinations of an output port
awaNameList	List of activity names
orderInputMode	Input port order mode for consuming tokens: *Iteration*, *Sequence*, *Any*
portState	State of the input and output ports: *Enable*, *Disable*, *EnableFeedback*
inputsToArgs	List of input port names for mapping *Task Arguments*
outputMode	Output port mode to issue tokens: *Single*, *Replicate*, *RoundRobin*
SendTo	List of of input port names destinations
params	List of activity *Parameters*
taskType	The qualified name of the Java class implementing the activity *Task*
tokenType	The qualified name of the Java class implementing the token type
ItersProposed	Set of iteration numbers proposed to achieve the reconfiguration agreement
AwaContext	Java Class for containing the *AWA Context*
awardConfigFile	Filename of the AWARD configuration file
wkfXmlFile	Filename of the XML workflow specification
newMaxIter	New maximum iteration number
resToOutputs	List of output port names for mapping *Task Results*

Figure 5.13: The signature of the operators provided by *DynamicAPI.jar*

5.7 The Components of the AWARD Machine

For supporting the execution of the *Autonomic Controller* of an autonomic workflow activity (AWA) the AWARD machine is implemented as a Java application named *AwaExecutor*. The implementation design follows the object-oriented programming model by defining Java classes to model data and the corresponding methods for implementing the algorithms and interactions between objects for supporting the AWARD machine. Therefore the relationships between the main internal components of the *AwaExecutor* are described using UML simplified diagrams to represent object classes and sequences of object interactions.

5.7.1 The Configuration of the Execution Environment

Considering that the multiple workflow activities (AWA) can be executed on distinct computing nodes the *AwaExecutor* requires configuration data related to the execution environment, for instance the location of the AWARD Space as well as the pathnames of directories containing the required files such as the workflow definition XML file. This configuration data is defined in a XML file as presented in Listing 5.10. It is assumed that each workflow activity executed by an *AwaExecutor* is able to read this configuration file, which should be replicated or accessible in all computing nodes.

Listing 5.10: The AWARD configuration environment file

```
1  ?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <AwardConfig>
3      <LogLevel>0</LogLevel>
4      <TimesExecEnabled>yes</TimesExecEnabled>
5      <JavaSDKbin>C:\Program Files\Java\jdk1.7.0_40\bin\java</JavaSDKbin>
6      <TSpacesHostTcpName>PLASS2</TSpacesHostTcpName>
7      <TSpacesTcpPort>8500</TSpacesTcpPort>
8      <TSpacesHostHttpName>PLASS2</TSpacesHostHttpName>
9      <TSpacesHttpPort>8501</TSpacesHttpPort>
10     <MainSpaceName>Tokens</MainSpaceName>
11     <AwaExecutorBaseDirectory>D:\AwaExecutor\</AwaExecutorBaseDirectory>
12     <ExecutableAwaExecutor>AwaExecutor.jar</ExecutableAwaExecutor>
13     <WorkflowDirectoryFiles>D:\WorkflowFiles\</WorkflowDirectoryFiles>
14 </AwardConfig>
```

The configuration file contains information related to the location (addresses and ports) of the AWARD Space, the name of the tuple space used, the home directories for getting the executable JAR of the *AwaExecutor* as well as the home directory to get the workflow specification. Additionally some configuration information is related to the support for monitoring the execution of workflows. The *AwaExecutor* produces log information according to levels where level 0 is the most verbose and level 10 only logs the current iteration number, the current state of the *State Machine*, the input and output tokens, the *Task Arguments* and the *Task Results*. The *AwaExecutor* also supports the

logging of the elapsed execution times for allowing the analysis of overheads or for instance to measure the benefits of applying reconfiguration scenarios in terms of execution times.

The first two entries in configuration file <*LogLevel*> and <*TimesExecEnable*> are useful for supporting monitoring and debugging of the workflows execution.

The *LogLevel* number is used by the AWARD machine components for producing more or less detailed logging information.

The *TimesExecEnable* flag is used to produce, or not, logging information related to the elapsed execution times of each internal step of the *Autonomic Controller*.

In Section 5.9 we demonstrate how this logging information is useful. This configuration information is accessed by all components of the *AwaExecutor* using the class represented in Figure 5.14.

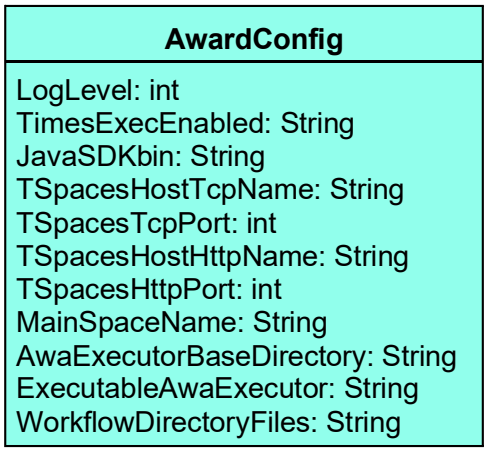


Figure 5.14: AWARD configuration class

5.7.2 Workflow Specification Classes

The workflow specification is obtained from the workflow specification XML file and is stored by a set of classes and the corresponding associations as represented in Figure 5.15. These classes store the details of a workflow specification as presented in Section 5.4 on page 152.

The associations between classes indicate: i) A workflow has one or more AWA activities; ii) An AWA activity has zero or more inputs, zero or more outputs, one *Task* and one control unit; iii) A *Task* is a software component; iv) The software component has one mapping from inputs to *Task Arguments* and a mapping from *Task Results* to outputs; v) Mapping arguments maps zero or more inputs to the *Arguments* list; vi) Mapping results maps zero or more outputs from *Task Results*.

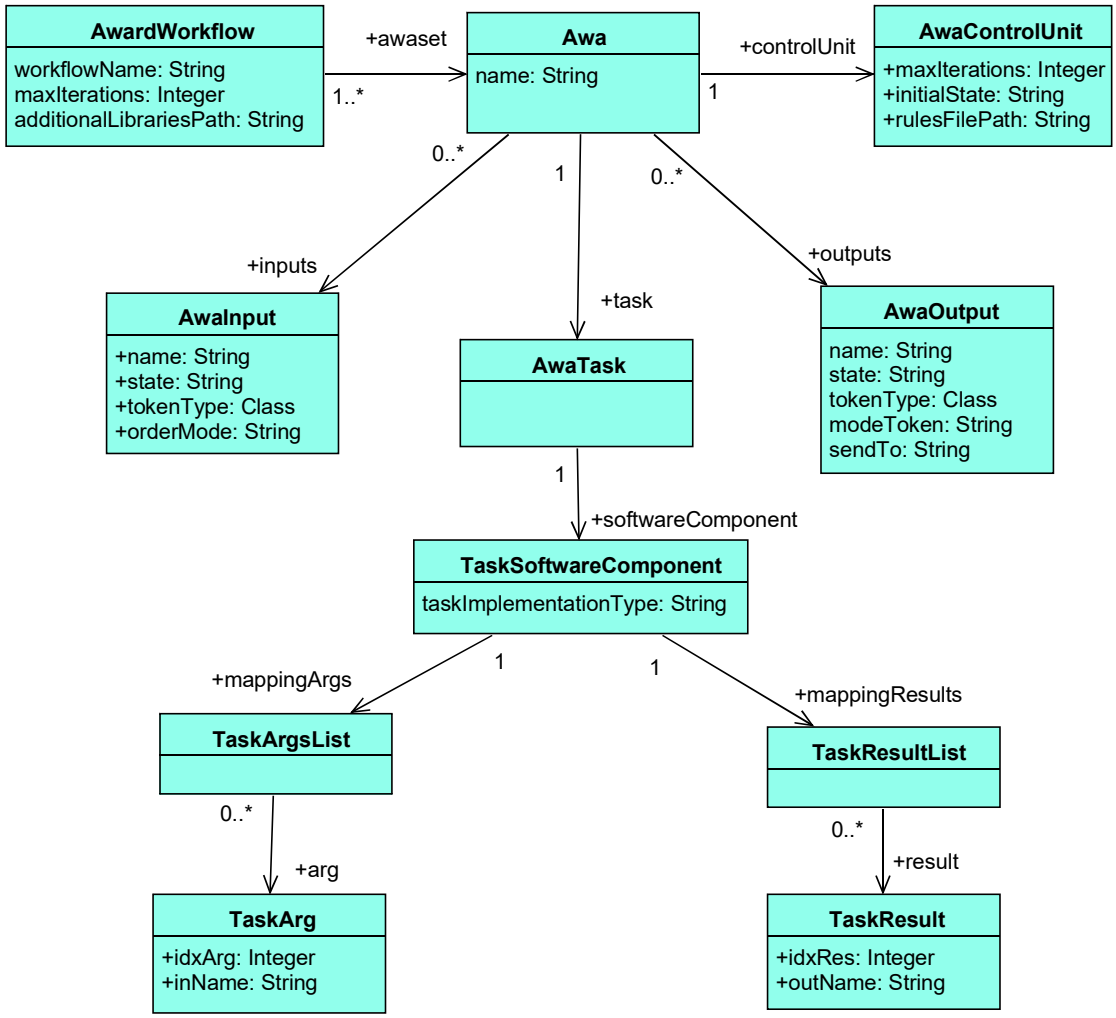


Figure 5.15: The workflow specification classes

5.7.3 The AWA Executor

The implementation of the AWA executor relies on the interactions between a set of software components modeled as Java objects. The main objects and their interactions are depicted in Figure 5.16 as a UML simplified sequence diagram that does not include the interactions related to handlers for getting the *AWA Context*, for performing dynamic reconfigurations as well as for forcing the AWA activity termination. These handlers will be presented in the Section 5.8. Therefore the main interactions are described in the following:

1. The main object named *AwaExecutor* is a Java application launched from tools developed for working with AWARD workflows;
2. The *AwaExecutor* object loads the AWARD configuration XML file;
3. The *AwaExecutor* creates the *AwaController* object responsible for the global coordination of the AWA activity life-cycle;

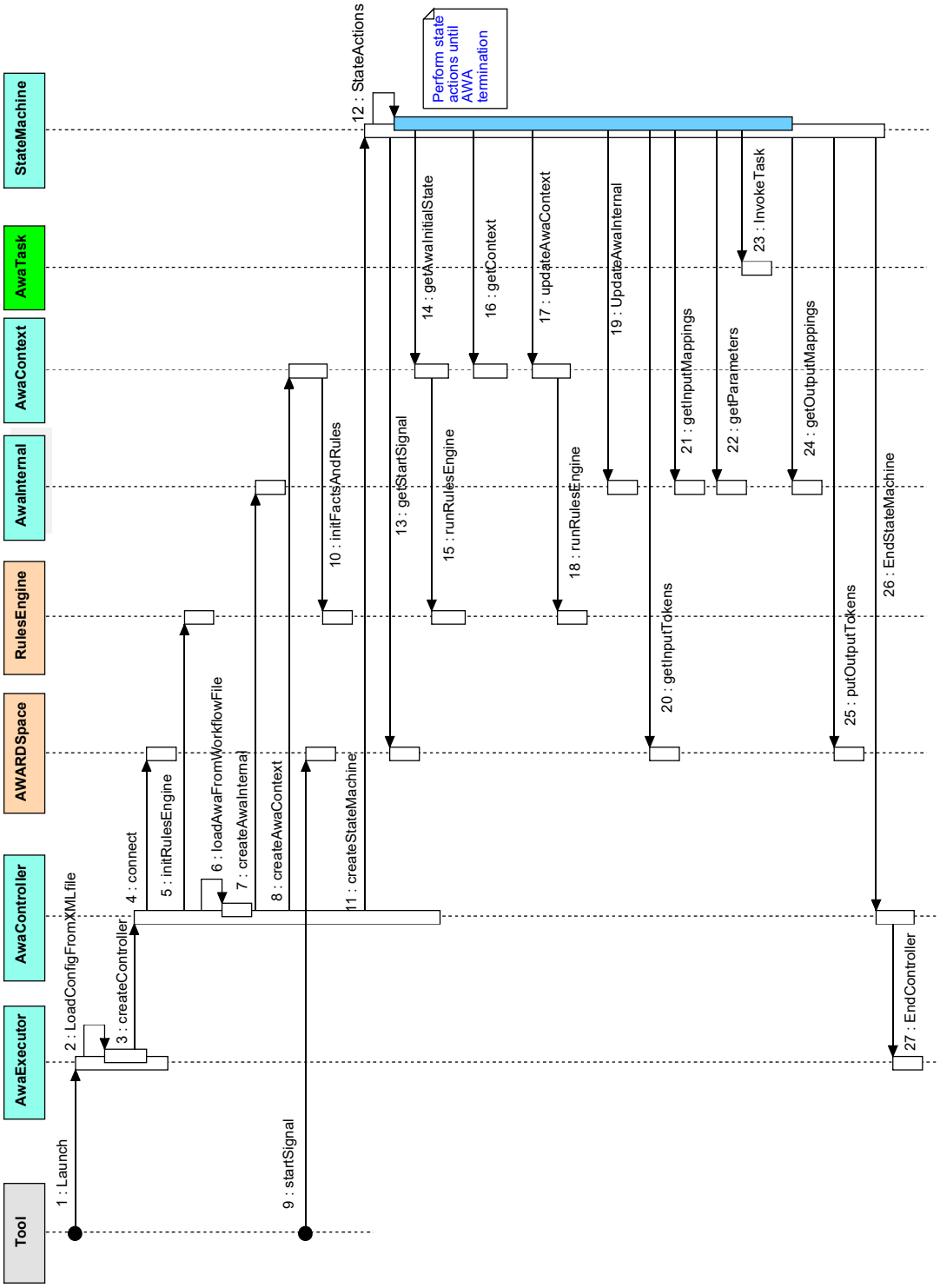


Figure 5.16: The sequence interactions of the AWARD machine components

4. The *AwaController* starts and establishes a connection to the AWARD Space;

5. The *AwaController* creates the *RulesEngine* object as an abstraction object for encapsulating the *Rules Engine* of the AWARD machine;

6. The *AwaController* loads the AWA activity specification from the workflow definition file;

7. The *AwaController* creates the *AwaInternal* object to store the global data related to the AWA activity specification, for instance the *Task* name, the input ports definitions and its corresponding mappings for *Task Arguments* as well as the *Task Results* and the corresponding mappings for the output ports;

8. The *AwaController* creates the *AwaContext* object for storing the internal *AWA Context* initially defined according to AWA activity specification and for reflecting all facts changed when rules fired on the *Rules Engine*;

9. The tool that launched the *AwaExecutor* (step 1), or another tool issues the start signal for this AWA activity. The signal can be issued in other sequence even before or after the creation of the AWA activity *State Machine*. This allows possible starting synchronization between distinct workflow activities launched in different moments. The start signal is simply implemented by injecting the *(awaName, "start")* tuple into the AWARD Space;

10. The *AwaContext* object initializes the *AWA Context* as facts and rules into the *Rules Engine*;

11. The *AwaController* creates the *StateMachine* object as the heart of the *Autonomic Controller* for controlling the execution of the AWA activity life-cycle;

12. The *StateMachine* begins the execution of its state actions;

13. In the *start* state the *State Machine* synchronizes by getting the start signal issued by any tool (step 9). The *State Machine* waits for the start signal until it has been issued by taking the *(awaName, "start")* tuple from the AWARD Space;

14. After synchronization in the *init* state the *StateMachine* object gets the next state for the *State Machine*. This state, specified in the workflow specification (*idle* or *WaitConfig* states) is stored into the *AwaContext* object (step 8) ;

15. The *AwaContext* always reflects the information existing as facts on the *Rules Engine*;

16. The *StateMachine* object gets the *AwaContext* information for performing the actions of each distinct state until the termination (*terminate*) state;

17. An important action performed by the *StateMachine* object consist of constant interactions with the *AwaContext* object for getting or updating the context information for instance the current state of the *State Machine* and the current iteration number or for changing the state of input and output ports as results of dynamic reconfigurations;

18. The *AwaContext* always reflects the information existing as facts on the *Rules Engine*;

19. Possible configuration changes resulting from dynamic reconfigurations are updated in the *AwaInternal* object;

20. In *input* state the *StateMachine* gets tokens from the AWARD Space for all input ports enabled;

21. In *mapIn* state the *StateMachine* gets the input mappings from the *AwaInternal* object;

22. In *invoke* state the *StateMachine* gets the *Task Parameters* from the *AwaInternal* object;

23. In *invoke* state the *StateMachine* creates the *AwaTask* object and invokes its entry point for executing the AWA *Task*;

24. When the code of the AWA activity *Task* returns, the *StateMachine* object gets the output mappings from the *AwaInternal* object;

25. In *output* state the *StateMachine* puts the output tokens into the AWARD Space;

26. When the *State Machine* ends, for instance by reaching the maximum number of iterations, it signals the *AwaController* object;

27. The *AwaController* terminates the *AwaExecutor* object which terminates the process launched for executing the autonomic workflow activity (AWA).

5.7.4 The Rules Engine

The term "rules engine" is sometimes ambiguous and used in multiple domains as a system that uses any form of rules that are applied to data to produce some outcomes, for instance, rules are used to define business logic, such as "A product order with total exceeding 100€ receives a 10% discount". However, the *Rules Engine* used in the *Autonomic Controller* is a *Production Rules Engine* [LGX10] based on the architecture depicted in Figure 5.17. *Facts* are stored into a *Working Memory* and *Rules* are stored in a *Production Memory*. The *Inference Engine* matches *Facts* against *Rules* and puts into the *Agenda* the *Rules* enabled to fire according to a pattern matching algorithm. The *Agenda* manages the execution of *Rules*. *Facts* in the *Working Memory* represent any type of data. A *Rule* is a two-part structure, *If <Conditions> then <Actions>*, evaluated as an if-then statement,

where *Conditions* are Boolean expressions based on *Facts* and *Actions* are assertions or retractions of *Facts* into the *Working Memory*. The pattern matching process uses a forward chaining algorithm, which is "data-driven" and reactive by allowing one or more *Rules* being concurrently enabled and executed for allowing the assertion or retraction of *Facts* into the *Working Memory*. For instance, the *Rete Match Algorithm* [Doo95; For90] is an efficient method for comparing a large collection of patterns to a large collection of objects and it is widely used in *Production Rules Engines*.

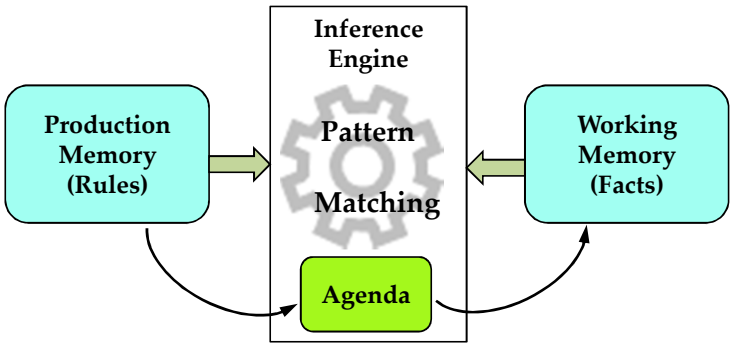


Figure 5.17: The Production Rules Engine architecture [LGX10]

✧ **JESS rules engine**

Despite the many implementations of *Production Rules Engine* available, the *Autonomic Controller* of the AWARD implementation is based on JESS [FH10] that uses the *Rete Match Algorithm* for processing rules. Besides JESS can be interactively used as a standalone program it also provides a flexible software library of small size (less than 1 MB), which can be embedded within any Java application.

The working memory of a JESS rule engine holds a collection of data organized as pure facts or shadow facts connected to the Java application objects for allowing the rules engine to know events that happened outside its working memory.

Every fact has a template with a name and a set of slots that is similar to a Java class or to a relational database table where slots are columns. Therefore a fact is like a single Java object or single row in a database table. As part of the *AWA Context* the current configuration of the activity input ports is managed by the *Rules Engine* of the *Autonomic Controller*. As an example, the Listing 5.11 presents the Java class used for representing the current state of an input port and the Listing 5.12 presents the corresponding JESS fact template and the addition (*assert command*) into the working memory of a particular fact for disabling an input port named *Awa1-In1*.

Similar to a basic programming statement *If (condition) then Action* a JESS rule has two parts, a left-hand side and a right-hand side, separated by *"=>"* symbol that can be read as *"then"*. The left-hand side part consists of patterns used to match facts into the working memory of the *Rules Engine*. The right-hand side part consists of a set of actions for changing the working memory.

A rule fires if the pattern matching is satisfied like the *Condition* is true in the *If* statement. Therefore rules can only react by addition and deletion of facts into the working memory. Identifiers starting with the question mark (?) character are variables names, for instance *?n*, *?f1*, *?ns* are valid variable names. The symbol "<-" is used to assign variables.

Listing 5.11: Java class to represent the input port state

```
package awaexecutor;

public class InputState {
  private String name;
  private String state;
}
```

Listing 5.12: A corresponding fact template and a single fact

```
(import awaexecutor.*)
(deftemplate InputState (declare (from-class InputState)))

;; add a single fact to the working memory
(assert (InputState (name Awa1-In1) (state Disable)))
```

As shown in the above Listing 5.12 the current state of input ports is managed by facts in the working memory that can be changed by rules. As an example, Listing 5.13 lists the rule named *ChangeInputState* used to change the state of any activity input port with the following behavior:

The rule fires when two facts in working memory match the pattern in the left-hand-side of the rule. Then the rule can be read as follows. For an input port named by *?n* with the current state identified by *?as* its state is changed to a new state identified by *?ns* when a fact named *ChangeInState* is inserted into the working memory and matches the pattern. The actions performed consist of deleting (*retract* command) the facts *?f1* and *?f2* that have fired the rule and by inserting (*assert* command) a new fact *InputState* for representing the new state *?ns* for the input port named by *?n*.

Listing 5.13: A generic rule for changing the state of an activity input port

```
(defrule ChangeInputState
   ?f1 <- (ChangeInState (name ?n) (astate ?as) (nstate ?ns))
   ?f2 <- (InputState (name ?n) (state ?as))
 =>
   (retract ?f1)
   (retract ?f2)
   (assert (InputState (name ?n) (state ?ns)))
)
```

The flexibility of the JESS API for Java programming did greatly simplify the implementation of the *Rules Engine* component of the *Autonomic Controller* of the AWARD machine. In fact, it is very easy and flexible to use a JESS rules engine for managing insertion

and deletion of facts, for defining and evaluating rules as well as for calling other useful functions.

The code developed for the *Autonomic Controller* relies on the JESS API for the Java programming. Some code snippets numbered from *CS1* to *CS5* are presented in Listing 5.14.

Listing 5.14: Code snippets illustrating the use of the JESS API

```
//CS1: Instantiate the Rules Engine
Jess.Rete engine = new Jess.Rete();

//CS2: Rules Engine initialization
engine.batch("InitialFactsAndRules.clp");

//CS3: Run the Rule Engine to fire enabled rules
engine.run();

//CS4: Insert a rule into the Rules Engine
engine.eval(
   "(defrule ChangeInputState
      ?f1 <- (ChangeInState (name ?n) (astate ?as) (nstate ?ns))
      ?f2 <- (InputState (name ?n) (state ?as))
      =>
      (retract ?f1)
      (retract ?f2)
      (assert (InputState (name ?n) (state ?ns)))
   )"
);
engine.eval(
  "(assert (ChangeInState (name Awa1-In1) (astate Disable) (nstate Enable)))"
);

//CS5: Function to get the names of the enabled input ports
ArrayList<String> getNamesOfInputPortsEnabled() {
  Jess.Context gctx=engine.getGlobalContext();
  Jess.Value inName, inState;
  ArrayList<String> list = new ArrayList<String>();
  Iterator it=engine.listFacts(); //iterate all facts in the working memory
  while (it.hasNext()) {
    Jess.Fact f = (Jess.Fact) it.next();
    if (f.getName().compareTo("InputState") == 0) { //if fact is InputState
      inName = f.getSlotValue("name"); //get the name of the input port
      inState = f.getSlotValue("state"); //get the state of the input port
      if (inState.symbolValue(gctx).compareTo("Enable") == 0) {//if state Enable
        list.add(inName.symbolValue(gctx)); //add the input name to the list
      }
    }
 }
 return list;
}
```

The *CS1* snippet shows how the object engine is instantiated from the class *Rete* of the JESS API. The *engine* object represents the abstraction of the *Rules Engine* globally inside the *Autonomic Controller*.

The *CS2* snippet shows how the *Rules Engine* is initialized from a text file using the JESS scripting language with facts and rules definition. For instance the fact template presented in Listing 5.12 and the rule presented in Listing 5.13 are initially loaded from a text file named *InitialFactsAndRules.clp* which is part of the predefined AWARD configuration.

Additionally, a workflow developer can customize the initial facts and rules by specifying a different file on the *ControlUnit* section of each AWA specification. This introduces great flexibility for allowing the internal context of distinct workflow activities to be initialized with different behaviors.

The *CS3* snippet forces the engine to run in order to fire all applicable rules.

The *CS4* snippet shows the *eval* function of the engine for inserting facts and rules into the *Rules Engine* object using text strings based on the JESS scripting language.

As examples, the rule *ChangeInputState* presented in Listing 5.13 is inserted into the *Rules Engine* as well as the insertion of a *ChangeInState* fact that fires the rule causing the input port named *Awa1-In1* to change its state from *Disable* to *Enable*.

The CS5 snippet shows the *getNamesOfInputPortsEnabled* function used by the *State Machine* to get the names of all enabled input ports. After getting the list of all facts from the engine the function iterates each fact named *InputState* and if the state value is equal to *Enable* the corresponding input port name is added to a list of names returned by the function.

5.8 The Handlers of the AWARD Machine

During the execution of an autonomic workflow activity (AWA) its *Autonomic Controller* is able to receive external asynchronous events published into the AWARD Space by any external tool developed in Java using the software library *DynamicLibrary.jar*. These events are related to: i) Requests for getting the *AWA Context*; ii) A command to explicitly and asynchronously force the AWA activity termination; and iii) the submission of dynamic reconfiguration operators.

According to the events functionality the dynamic library publishes tuples into the AWARD Space for notifying the corresponding handlers previously registered by each AWA activity in the AWARD Space.

For subscribing and processing the asynchronous events from the AWARD Space the AWARD machine implements three notification handlers described in the following sections.

5.8.1 The Request Context Handler

The *Autonomic Controller* implements the *ContextHandler* for handling the events related to requests for getting *AWA Context*. As illustrated in Figure 5.18 the sequence of actions to get the *AWA Context* is described in the following:

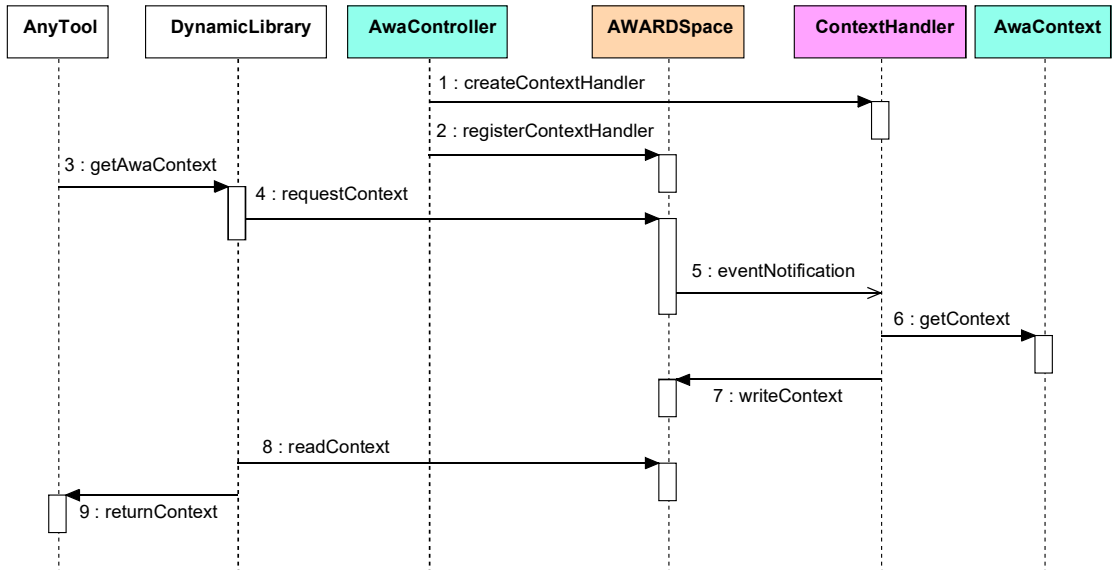


Figure 5.18: The sequence of interaction actions to get the *AWA Context*

1. In the initializing phase of the Autonomic Controller, the AwaController creates the ContextHandler object for processing the future requests of the *AWA Context*;
2. The AwaController registers the ContextHandler object on the AWARD Space to enable future notifications;
3. Any tool (AnyTool) developed in the Java language, for instance a tool for monitoring the workflow execution calls the getAwaContext function of the DynamicLibray library for requesting the *AWA Context*;
4. This getAwaContext function injects a tuple into the AWARD Space containing the request for the *AWA Context* and blocks until it receives a reply;
5. By asynchronous notification the request tuple is received by the ContextHandler;
6. The ContextHandler collects the context information from the AwaContext object;
7. The context information is written as a tuple into the AWARD Space for satisfying the blocked request performed in action 4;
8. The getAwaContext function reads the *AWA Context* from the AWARD Space ;
9. The context information is returned to the tool.

5.8.2 The Explicit Forced Termination Handler

During the execution of a workflow unexpected situations, for instance, the failure of an upstream activity introduces the need for ensuring that the termination of an AWA activity is always possible. The *Autonomic Controller* has a *Forced Termination Handler* for accepting a command to explicitly and asynchrounouly terminate the activity independently of its internal context. As illustrated in Figure 5.19 the sequence of actions to force the AWA activity termination is described in the following:

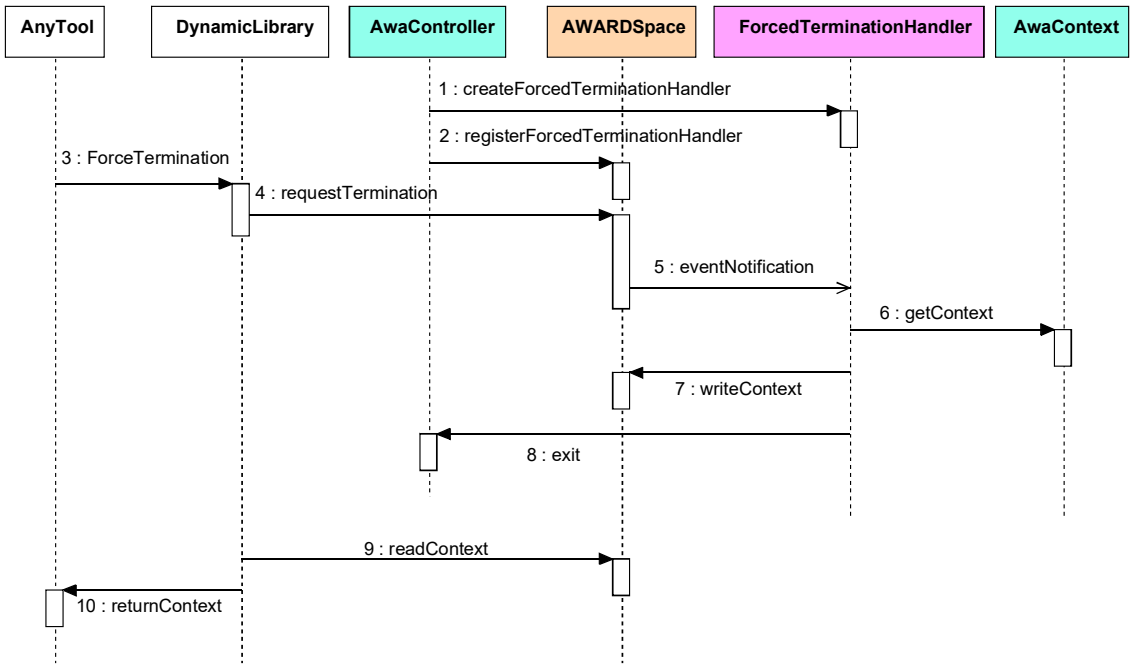


Figure 5.19: The sequence interactions to force an AWA termination

1. In the initializing phase of the *Autonomic Controller*, the *AwaController* creates the *ForcedTerminationHandler* object for processing a future command to force the AWA activity termination;

2. The *AwaController* registers the *ForcedTerminationHandler* object on the AWARD Space to enable the forced termination notification;

3. Any tool (*AnyTool*) developed in Java language, for instance the *KillAwa.jar* AWARD tool, presented in Section 5.9, calls the *ForceTermination* function of the *DynamicLibray* to request the AWA termination;

4. The *ForceTermination* function injects a tuple into the AWARD Space containing the request for AWA activity termination and blocks until receives a reply;

5. By asynchronous notification the request tuple is received by the *ForcedTerminationHandler*;

6. The *ForcedTerminationHandler* collects the context information from the *AwaContext* object;

7. The context information is written as a tuple into the AWARD Space for satisfying the blocked request performed in action 4;

8. The *ForcedTerminationHandler* forces the *AwaController* to exit;

9. The *ForceTermination* function reads the *AWA Context* from the AWARD Space;

10. The context information is returned to the invoking tool (*AnyTool*).

5.8.3 The Dynamic Reconfiguration Handler

The Autonomic Controller has a *Dynamic Reconfiguration Handler* (DRH) for subscribing the asynchronous events related to dynamic reconfiguration operators. As illustrated in Figure 5.20 the sequence of interactions for performing dynamic reconfigurations is described in the following:

1. In the initializing phase of the *Autonomic Controller*, the *AwaController* creates the *ReconfigurationHandler* object;

2. The *AwaController* registers the *ReconfigurationHandler* object on the AWARD Space for subscribing asynchronous events related to dynamic reconfigurations. These events are represented by tuples *(RID*, *Event*, *AWAname*, *argsList)*, where *RID* is a reconfiguration plan identifier, *Event* identifies the event associated to the reconfiguration operator, *AWAname* is the activity name, and *argsList* is a list of arguments according to the event;

3. By integrating the *DynamicLibray* any Java application as a tool can submit reconfiguration plans. The tool starts a reconfiguration plan by calling the *BeginReconfiguration* library function for getting a global unique reconfiguration identifier (*RID*) (steps 4, 5 and 6). The uniqueness of the *RID* identifier is ensured by the AWARD Space that stores a tuple with the *(RID*, *value)* template;

4. The implementation of the *getRID* request gets the tuple from the tuple space, increments the value of *RID* and puts the new tuple into the AWARD Space. The atomicity of retrieving a tuple from the AWARD Space ensures the uniqueness of the *RID* identifier;

5. The *getRID* request returns the next *RID* reconfiguration identifier;

6. The new *RID* is returned to the invoking tool;

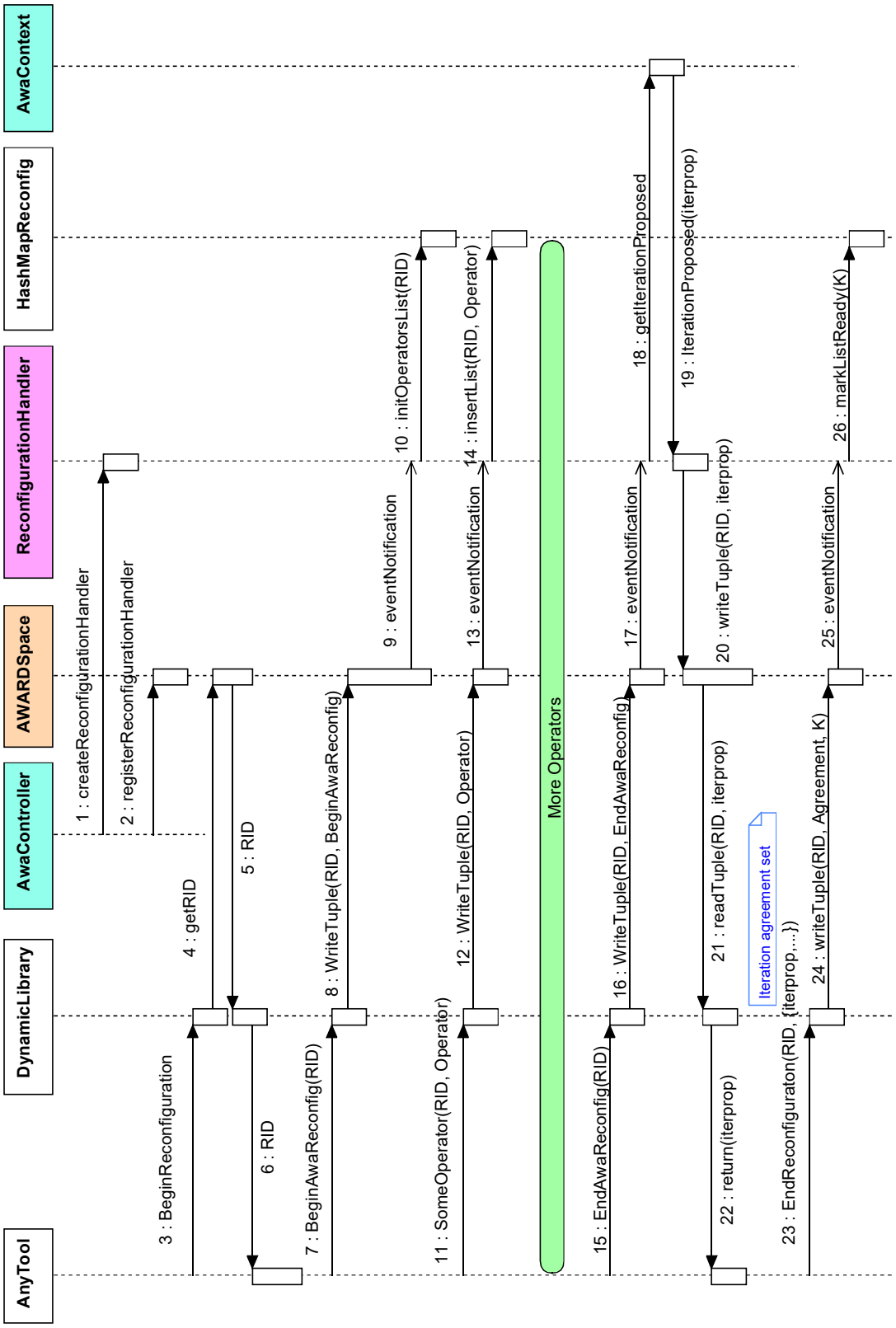


Figure 5.20: The sequence of interactions for supporting dynamic reconfigurations

7. For each activity involved in the reconfiguration plan there is a reconfiguration block started and ended respectively by *BeginAwaReconfig* and *EndAwaReconfig* commands (step 15). This block can apply a sequence of operators illustrated by the *SomeOperator* generic operator (step 11);

8. The implementation of the *BeginAwaReconfig* command consists of writing a tuple in the AWARD Space for enabling the asynchronous notification of the reconfiguration handler;

9. The *ReconfigurationHandler* is notified and consumes the *BeginAwaReconfig* operator;

10. The *ReconfigurationHandler* initializes a new entry identified by *RID* into the hash table *HashMapReconfig* to store the following sequence of operators;

11. The tool submits the *SomeOperator* operator;

12. The implementation of all operators is similar by writing a tuple containing the operator;

13. The *ReconfigurationHandler* is notified that there is a new operator;

14. The *ReconfigurationHandler* reads and inserts the operator into the list on the *HashMapReconfig* hash table;

15. After the tool has submitted all operators involved in the reconfiguration plan, it submits the *EndAwaReconfig* command;

16. The implementation of the *EndAwaReconfig* command writes the respective tuple and blocks, waiting for an iteration number proposed by the AWA activity to submit its part of the reconfiguration plan;

17. The insertion of the *EndAwaReconfig* tuple notifies the *ReconfigurationHandler*;

18. The *ReconfigurationHandler* requests the proposed iteration number to the *AwaContext* object;

19. The *ReconfigurationHandler* gets the adequate proposed iteration number (*iterprop*) from the *AwaContext* object. This is the current iteration number plus one for indicating that from the point of view of this AWA activity, the dynamic reconfiguration can take place as soon as possible, that is at the next iteration, or the value -1 if the reconfiguration can not be applied, for instance the activity is executing its maximum iteration number;

20. The proposed iteration number (*iterprop*) returned by the *AwaContext* is written as a tuple in the AWARD Space by the *ReconfigurationHandler*;

21. The implementation of the *EndAwaReconfig* command unblocks by reading the tuple with the proposed iteration number (*iterprop*). This iteration number is returned by the *EndAwaReconfig* command;

22. The tool *AnyTool* that has submitted the reconfiguration plan stores the iteration proposed (*iterprop*) in the iteration agreement set that is passed to the *EndReconfiguration* command;

23. The implementation of the *EndReconfiguration* command calculates the maximum iteration contained in the iteration agreement set as a value *K*, which is the iteration agreed upon where all activities involved will apply its part of the reconfiguration plan;

24. The iteration agreement is written as a tuple in the AWARD Space;

25. The iteration agreement tuple notifies the reconfiguration handlers of all activities involved in the reconfiguration plan that an agreement is achieved to be applied in the *K* iteration;

26. If *K* is equals to -1, the reconfiguration plan was not committed and the reconfiguration sequence marked with *RID* in the *HashMapReconfig* hash table is discarded by the *ReconfigurationHandler*; otherwise the *ReconfigurationHandler* marks, in the *HashMapReconfig* hash table, that the *RID* reconfiguration plan with a list of operators is ready to be applied at the *K* iteration.

✧ **Processing reconfiguration plans**

Before starting each iteration the *State Machine* verifies in the *HashMapReconfig* hash table if there are any reconfiguration plans ready to be applied. As illustrated in Figure 5.21 the sequence of interactions for processing dynamic reconfiguration operators is described in the following:

1. The *State Machine* in the *idle* state checks if the current iteration number is equal to any *K* iteration number resulting from previous agreements;

2. If some ready reconfiguration exists, then the *State Machine* goes to the *Config* state;

3. In *Config* state the *State Machine* processes the reconfiguration plan by processing the list of dynamic reconfiguration operators;

4. The current AWA activity configuration is changed in the *AwaInternal* object;

5. After processing all operators of the reconfiguration plan the *State Machine* comes back to the *idle* state to proceed with the *K* iteration number, where the AWA activity has a new structure and a new behavior according to the reconfiguration plan. As an example, if the reconfiguration changed the activity *Task* then in the *invoke* state the *State Machine* instantiates the new *Task* object and invokes its entry point.

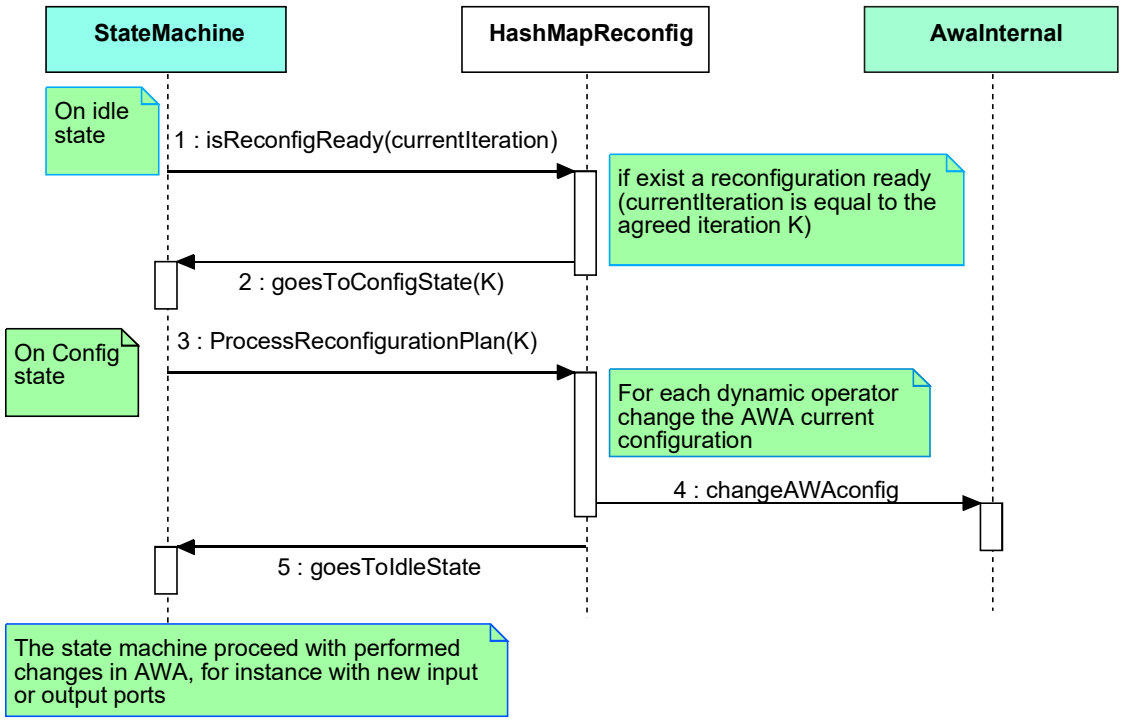


Figure 5.21: Processing reconfiguration plans

5.9 The AWARD Tools

The main requirement for developing the AWARD tools was producing a set of flexible commands that can be easily used by workflow developers for allowing the execution of AWARD workflows. Therefore a minimal but sufficient set of tools are provided by the AWARD framework for supporting launching the AWA activities of AWARD workflows and for allowing the observation of logging information in order to monitor the execution of the long-running AWARD workflows.

According to the application algorithms, data and resources location, the workflow developers define partitions of AWA activities and assign these partitions to the available computational nodes. Thus the AWARD framework provides commands to launch one or more AWA activities as Java applications (one operating system process per each AWA activity) on any computing node, whether it is local or remote. In fact, using any *Secure Shell* (SSH) client it is possible launching AWA activities on virtual machines on cluster or cloud infrastructures.

The autonomic characteristics of the AWARD workflows allow activities to be executed separately on distinct computing nodes and subject to dynamic reconfigurations.

According to the elapsed execution time of the *Task* algorithm, each activity produces output tokens at its own pace and consequently the connected downstream activities also consume these tokens at their own paces. This requires the capability of dynamically monitoring the state of the execution of the workflow activities.

The AWARD framework uses a logging mechanism supported by the AWARD Space given that it is shared by all AWA activities. For instance, when the *State Machine* of an AWA activity reaches a faulty state, the *AWA Context* is saved in the AWARD Space as logging information (actions a_8, a_9 and a_{10} of Table 3.4 on page 83).

This logging functionality of the AWARD machine is an important characteristic for workflow developers to perform debugging actions. During the prototype implementation this characteristic was a fundamental aspect for detecting and correcting software bugs, as well as for easing the experimentation with implementation alternatives.

5.9.1 Basic Tools for Launching and Monitoring the Workflow Execution

The developed and provided set of AWARD tools, presented in Table 5.2, includes tools for launching the AWARD Space server, for launching activities, for getting the logging information, for instance, the elapsed execution time of an activity *Task*. These tools can be executed as shell commands or easily integrated within any specific problem domain tool. A user can develop other tools in order to manage the execution of AWARD workflows on distributed infrastructures. As an example, this was done in the context of the development of a data analytics application expressed as a distributed AWARD workflow [GAC12]. In Chapter 6 we present use cases where tools for launching activities are integrated with other tools to perform text mining experiments as well as a real scenario where the logging information is used to detect failures on activity *Task* invocation followed by a recovery using a dynamic reconfiguration plan for changing the activity *Task*.

The tools presented in Table 5.2 are using the following arguments:

- *[tcpport=8500 httpport=8501]*: Redefinition of the TCP/IP ports to be used by the AWARD Space server;
- <*Config File*>*.xml*: The absolute name of the AWARD configuration file for defining the execution environment for instance, the pathname for the adequate version of the Java platform, file system directories for workflow files and *Task* libraries, as well as the location of the AWARD Space;
- <*workflow File*>*.xml*: The filename of the file that contains the workflow specification. This filename is appended to the directory <*WorkflowDirectoryFiles*> specified in the AWARD configuration file;
- <*AWA name*>: An AWA activity name of the workflow activities;
- <*Dump text file*>*.txt*: The filename of a text file for dumping logging information used for monitoring the workflow execution. This filename is appended to the directory <*WorkflowDirectoryFiles*> specified in the AWARD configuration file.

The AWARD tools are described in the following:

Table 5.2: The developed and provided AWARD tools

Tool: *Launch the AWARD Space server* Usage: java -jar AwardSpace.jar [tcpport=8500 httpport=8501]
Tool: *Launch an AWA executor as a java application* Usage: java -jar AwaExecutor.jar <Config File>.xml <workflow File>.xml <AWA name>
Tool: *Launch workflow partitions with one or more AWA* Usage: java -jar AwardLaunchAWA.jar <Config File>.xml <workflow File>.xml <AWA name1> ... <AWA nameN>
Tool: *Launch workflow partitions with one or more AWA where the AWA executor waits for a signal to start* Usage: java -jar AwardLaunchAWAwaiting.jar <Config File>.xml <workflow File>.xml <AWA name1> ... <AWA nameN>
Tool: *Send a signal for starting AWA executors of a workflow partition with one or more AWA* Usage: java -jar AwardAWAstart.jar <Config File>.xml <workflow File>.xml <AWA name1> ... <AWA nameN>
Tool: *Launch a complete workflow (all AWA) in the same computing node* Usage: java -jar AwardLaunchWkf.jar <Config File>.xml <workflow File>.xml
Tool: *Forces the termination of an AWA* Usage: java -jar KillAwa.jar <Config File>.xml <AWA name>
Tool: *Dump the logs of an AWA* Usage: java -jar DumpLogs.jar <Config File>.xml <AWA name> <Dump text file>.txt
Tool: *Dump the elapsed execution times of an AWA* Usage: java -jar DumpExecTimes.jar <Config File>.xml <AWA name> <Dump text file>.txt
Tool: *Gets the context of an AWA activity* Usage: java -jar GetContext.jar <Config File>.xml <AWA name> <Dump text file>.txt

✧ **Tool for launching the AWARD Space**

- *AwardSpace.jar*: This tool is used to launch the AWARD Space.

✧ **Tools for launching workflow activities**

- *AwaExecutor.jar*: This tool is used to launch the executor of an autonomic workflow activity (AWA) as a Java application. The tool parses the XML workflow specification file for searching the AWA activity name passed as an argument. After initializing the execution environment this tool starts the *State Machine* in the *init* state to init the activity life-cycle;

- *AwardLaunchAWA.jar*: This tool allows launching a workflow partition formed by one or more AWA activity executors on the same computing node. Each executor is launched as an independent Java application. In this way one workflow can be spread on multiple computing nodes on a cluster or on a cloud infrastructure;

- *AwardLaunchAWAwaiting.jar*: This tool is similar to the *AwardLaunchAWA.jar* but the AWA activity executors block in the *start* state, waiting for a start signal that must be issued by the *AwardAWAstart.jar* tool;

- *AwardAWAstart.jar*: This tool issues one or more starting signals for one or more AWA activity executors launched by the *AwardLaunchAWAwaiting.jar* tool;

- *AwardLaunchWkf.jar*: This tool launches all AWA activities of a workflow on the same computing node.

✧ **Tool for explicitly and asynchronously forcing an activity termination**

- *KillAwa.jar*: This tool is used for forcing the termination of a workflow activity independently of its internal state. The tool simply invokes the *ForcedTermination* operator for activating the corresponding handler that forces the exit of the *AwaExecutor*.

✧ **Tools for monitoring workflow activities**

- *DumpLogs.jar*: This tool extracts, from the AWARD Space, the logging information related to the AWA activity whose name is passed as an argument. The information is stored in the text file whose name is passed as an argument;

- *DumpExecTimes.jar*: This tool extracts from the AWARD Space the logging information containing the elapsed execution times of an AWA activity whose name is passed as an argument. The information is stored in the text file whose name is passed as an argument;

- *GetContext.jar*: This tool is used for getting the *AWA Context* for monitoring the AWA activity internal state. The tool simply invokes the *GetAwaContext* operator provided by the *DynamicLibrary* API for activating the corresponding reconfiguration handler.

✧ **The logging information details**

The *Autonomic Controller* of each AWA activity stores tuples into the AWARD Space containing logging information related to levels of the current execution context. The workflow developers can configure the logging level using the <*LogLevel*> field of the AWARD configuration file as a number between 0 and 10. The level 0 is the most detailed, for instance it includes all actions executed in each state of the *State Machine*, such as the state of input and output ports, the current iteration and the values of the input and output tokens as well as the *Task Arguments*. A level greater than 0 excludes some details, for instance level 5 only produces logging information related to the *Task* invocation, including the corresponding *Arguments* and *Results* or exceptions thrown by the software component that implements the *Task* algorithm. The level 10 only produces

Listing 5.15: An example of logging information

```
("Log:", "0", "Add", "Dir=D:\AWARD\WorkflowFiles\, File=workfAddIntegers.xml")
("Log:", "0", "Add", "Workflow Name:workflow Add two Integers")
("Log:", "0", "Add", "NIterations=20")
("Log:", "0", "Add", "AdditionalLibs=D:\AWARD\TaskLibrary\AwardTaskLib.jar")
("Log:", "0", "Add", "AWA specification")
("Log:", "0", "Add", "Maximum iterations: Not redefined")
("Log:", "0", "Add", "Initial state: Idle")
("Log:", "0", "Add", "RulesFileName=TaskBasicRules.clp")
... (Omitted other AWA specification details)
("Log:", "2", "Add", "State Machine initialization state :init")
("Log:", "2", "Add", "goes to state:idle:Iteration:0")
... (Omitted details of iterations 0 until 14)
("Log:", "2", "Add", "Current Iteration:15")
("Log:", "2", "Add", "goes to state:idle")
("Log:", "2", "Add", "Check for Reconfigurations isReconfReady:FALSE")
("Log:", "2", "Add", "goes to state: input at iteration=15")
("Log:", "2", "Add", "Input Enable:AddI1")
("Log:", "2", "Add", "Input Enable:AddI2")
("Log:", "0", "Add", "Input:AddI1 Tuple read:(15, 0, "AddI1", 25)")
("Log:", "0", "Add", "Input:AddI2 Tuple read:(15, 0, "AddI2", 30)")
("Log:", "2", "Add", "Token iteration:AddI1:15")
("Log:", "2", "Add", "Token iteration:AddI2:15")
("Log:", "2", "Add", "Token sequence:AddI1:0")
("Log:", "2", "Add", "Token sequence:AddI2:0")
("Log:", "2", "Add", "Token value:AddI1:25")
("Log:", "2", "Add", "Token value:AddI2:30")
("Log:", "5", "Add", "goes to :mappingInputs at iteration=15")
("Log:", "5", "Add", "idxArg=0:AddI1=25, idxArg=1:AddI2=30")
("Log:", "5", "Add", "goes to:Invoke:awardtasklib.TaskIntegerAdd:iteration=15")
("Log:", "5", "Add", "Invoke task with args:25, 30")
("Log:", "5", "Add", "Task Results: idxRes=0:55")
("Log:", "5", "Add", "goes to :MapOutputs at iteration=15")
("Log:", "5", "Add", "idxRes=0:AddO1")
("Log:", "5", "Add", "mappingResults to:AddO1:Enable:Replicate:SendTo->InI1")
("Log:", "2", "Add", "Token value:AddO1:55")
("Log:", "2", "Add", "goes to : Output at iteration=15" )
("Log:", "0", "Add", "Output:AddO1 tuple write:(15, 0, "InI1", 55)")
("Log:", "2", "Add", "insert fact IncrementCurIteration on JESS")
("Log:", "2", "Add", "goes to :Idle at iteration=15")
("Log:", "2", "Add", "Run the rules on JESS")
("Log:", "2", "Add", "Current Iteration:16")
... (similar tuple sequences until iteration 19)
("Log:", "2", "Add", "goes to :Idle at iteration=19")
("Log:", "2", "Add", "Run the rules on JESS")
("Log:", "2", "Add", "Current Iteration:20")
("Log:", "2", "Add", "Awa Executor terminates")
```

logging information related to unexpected exceptions, for instance, when accessing the AWARD Space and the JESS rules engine.

Therefore AWARD workflow developers can monitor the workflow execution by continuous observation of the logging tuples into the AWARD Space using any HTTP browser that connects to the AWARD Space server, even after the workflow was terminated. Before the AWARD Space server shutdown, the logging information can be retrieved and stored in a text file using the AWARD tool *DumpLogs.jar* for allowing future analysis.

To illustrate an example of the logging functionality consider a workflow with an activity named *Add* that adds two integer numbers received at two input ports named *AddI1* and *AddI2*. The addition result is sent to an output port named *AddO1* connected to an activity with an input port named *InI1*.

Listing 5.15 partially shows the logging information produced by considering the number 20 as the maximum number of iterations. The first log entries partially report the *Add* activity specification details. Afterwards the *State Machine* reports its initialization state by indicating that the AWA activity starts execution on *idle* state at iteration 0.

The logs for the 15^{th} iteration are shown in detail, where we can observe all actions of the *State Machine* including the *Arguments* and *Results* of the *Task* invocation. Finally the logs of the 19^{th} iteration are presented to indicate the AWA activity termination when it reaches the maximum iteration number.

For analyzing the workflow performance in terms of the elapsed execution times the AWARD machine has the possibility to produce logging of the execution times of each step of the *State Machine*. This allows analysis of overheads, for instance getting answers to the following questions, such as, *"What are the slower activities?"*, *"What is the overhead for getting tokens from the AWARD Space?"*, *"What is the average time for executing an iteration?"*, among others. This functionality can be enabled or disabled through the *<TimesExecEnabled>* field of the AWARD configuration file. When the functionality is enabled the state machines of all activities also write log tuples into the AWARD Space that can be monitored during workflow execution or extracted using the *DumpExecTimes.jar* tool. The elapsed execution times are based on the system time of the computer where the AWA activity is running. In case of multiple computing nodes for executing the multiple workflow activities we assume that the possible clock drifts are negligible.

As an example the elapsed execution time of the 15^{th} iteration of the activity *Add*, which adds two integer numbers from its two input ports and produce on its output port the result, is presented in Listing 5.16.

Listing 5.16: Example of logging with elapsed execution time

```
1 ("TIMES:", "Add", "15", "BI-1443376098855", "AI-1443376098948",
2                     "BT-1443376098962", "AT-1443376098966",
3                     "BO-1443376098995", "AO-1443376099066")
```

The log presents the current system time in milliseconds for the following steps of the *State Machine*: *Before input* (BI); *After input* (AI); *Before Task* (BT); *After Task* (AT);

Before Output (BO) and *After Output* (AO). The log of Listing 5.16 allows to calculate the following elapsed execution times:

- 93 milliseconds (1443376098948-1443376098855) as the difference between AI and BI to get the input ports tokens;
- 71 milliseconds (1443376099066-1443376098995) as the difference between AO and BO to put the output port token;
- 9 milliseconds (1443376098966-1443376098962) as the difference between AT and BT as the execution time of the *Task*.

Furthermore we can roughly calculate the elapsed iteration execution time of 211 milliseconds as the difference between AO and BI (1443376099066-1443376098855).

5.9.2 A Graphics Interface Tool for Managing AWARD Workflows

Additionally for executing AWARD workflows on standalone computers the AWARD framework also provides a basic tool with a graphical user interface presented in Figure 5.22.

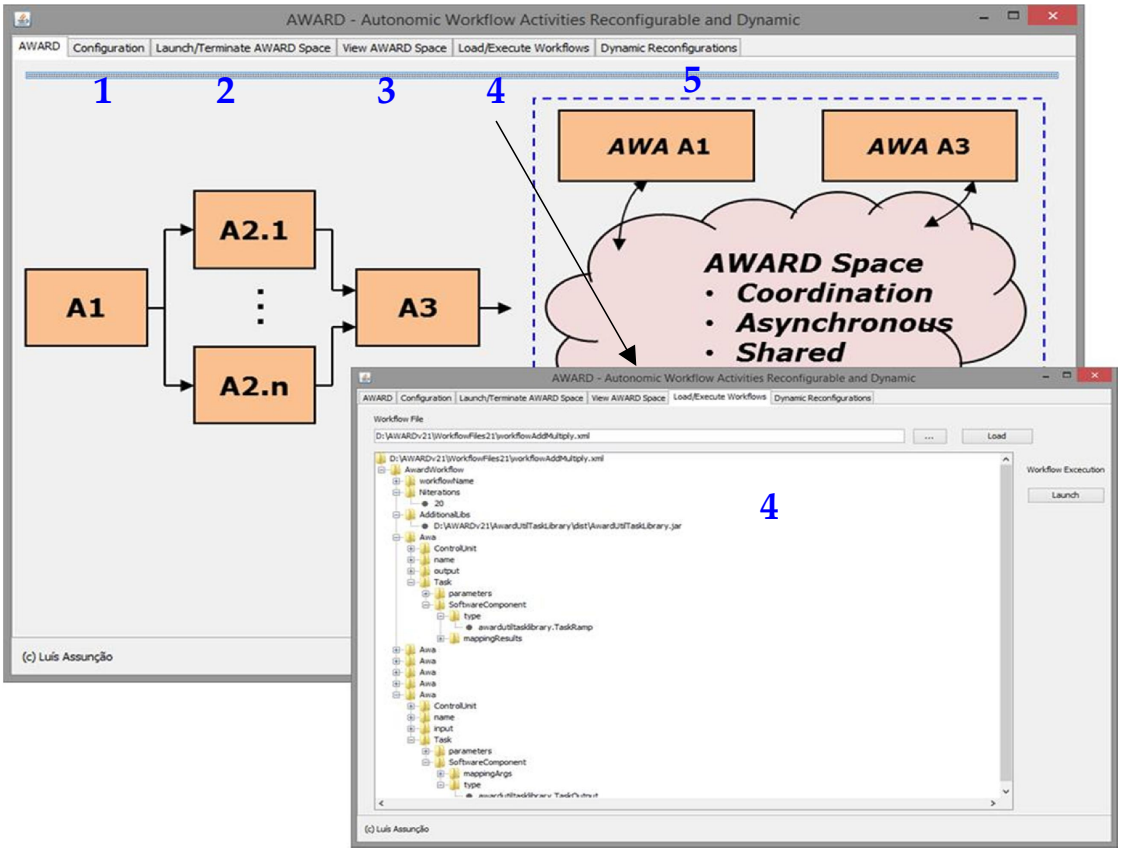


Figure 5.22: *AwardGUI.jar* tool to execute AWARD workflows on standalone computers

The tool shown in Figure 5.22 named *AwardGUI.jar* allows:

1. View and change the AWARD configuration file (Menu 1);
2. Launch or terminate the AWARD Space server (Menu 2);
3. Visualize/Monitor the current content of the AWARD Space (Menu 3);
4. Load and visualize an AWARD workflow specification as well as launch all activities of the workflow for execution (Menu 4, which opens the dialog box to navigate the workflow specification);
5. Submit basic but useful reconfiguration plans, such as change *Parameters* or change the *Task* of the running activities (Menu 5).

5.10 Chapter Conclusions

The architecture design and implementation of an operational working prototype to support the development, execution, monitoring and dynamic reconfiguration of AWARD workflows follows the object-oriented programming paradigm using the JAVA language.

In addition to the native Java environment, the AWARD framework resulting from this implemented prototype has minimum dependencies upon third-party software components, namely it depends on the Java JESS library to implement the *Rules Engine* and the Java IBM TSpaces Server library for implementing the AWARD Space.

The AWARD framework developed, including the AWARD tools and the Dynamic library for performing dynamic reconfigurations is very lightweight and so it provides great flexibility for running AWARD workflows on very distinct types of computing nodes, such as, standalone computers as well as virtual machines on cluster and cloud infrastructures. As two examples the Java executable, including the necessary libraries, for executing an AWA activity has a size less than 7 MB and the AWARD Space server, as a standalone Java application server, has a size less than 1 MB.

A useful characteristic of the AWARD framework is the possibility (by configuration) to explore logging information during the workflows execution. For long-running workflows the observation of this logging information is a crucial point to monitor the behavior of all workflow activities even those that are running on distinct computing nodes. In addition this logging information is also very helpful for code debugging. This was very important during the prototype implementation but it was also shown important when used by a workflow developer to perform application-level debugging.

Chapter 6

Evaluation of the AWARD Model and its Implementation

Evaluation of the AWARD model and its implementation to develop concrete workflow scenarios and application cases.

This chapter discusses the evaluation of the AWARD model and its implementation concerning its functionality, expressiveness and feasibility for developing scientific workflows. The evaluation explores the AWARD characteristics for operational workflow execution on parallel and distributed infrastructures, for supporting dynamic workflow reconfigurations and the feasibility of using the implemented AWARD framework for developing concrete application cases.

In Section 6.1 we describe the strategy followed for evaluating the AWARD functionality, expressiveness and feasibility for developing workflows using scenarios arranged in three dimensions that are described in the remaining sections.

In Section 6.2 the dimension of parallel and distribution execution includes scenarios to demonstrate how basic and non basic workflow patterns can be executed using AWARD where all activities or activity partitions can be mapped for execution on a single computer or on distributed infrastructures. This section also presents a discussion related to execution performance and the subjacent overheads.

In Section 6.3 we discuss a set of structural and behavioral workflow reconfiguration scenarios for evaluating the AWARD characteristics to support the dimension of dynamic workflow reconfigurations using the reusable software library (*DynamicLibrary.jar*), which provides a set of reconfiguration operators.

In Section 6.4 we demonstrate the feasibility of using AWARD for developing a set of concrete application cases, such as using AWARD for implementing the MapReduce model, recovering from faulty cloud services, and a text mining application involving a considerable number of activities and virtual machines for their execution which actually executed on local clusters and on two different public cloud infrastructures: Amazon AWS [Ama15b]; and Lunacloud [Lun15].

In Section 6.5 we compare AWARD with other workflow systems in particular Triana [Chu+06] and Kepler [Kep13], and we discuss the autonomic characteristics of the AWARD model.

Finally in Section 6.6 we present the conclusions and the lessons learned from the experiments for evaluating the AWARD model and its implementation.

6.1 The Strategy for Evaluating the AWARD Framework

The strategy for evaluating the AWARD model was guided by the motivations (Chapter 1, Section 1.1) and the rationale for developing the AWARD model (Chapter 3, Section 3.1), including the support for dynamic reconfigurations (Chapter 4, Section 4.1).

Bearing always in mind the operationality of the implemented architecture and presenting a comparison with other existing scientific workflow systems, the evaluation of the AWARD framework aims to demonstrate three fundamental AWARD characteristics: i) The functionality; ii) The expressiveness; and iii) The feasibility for developing scientific workflows.

As illustrated in Figure 6.1 these AWARD characteristics are evaluated following three dimensions:

1. The AWARD support for parallel and distributed workflow execution;
2. The AWARD support for dynamic workflow reconfigurations;
3. The feasibility of using AWARD for implementing application cases.

The above three dimensions are detailed in the following:

✧ **Parallel and distributed execution**

According to this dimension the AWARD model is evaluated in its capability for specifying scientific workflows with multiple activities connected using basic and non-basic patterns [Aal+00b]. We consider some basic patterns, such as Sequence (an activity is enabled after the completion of another), Parallel split (a single activity splits into multiple activities which can be executed in parallel), and Join (joining of multiple parallel activities). As non-basic patterns we consider the Feedback loop structural pattern (activities receive tokens and synchronize with outputs from the previous iteration) and the Multi-choice or Conditional routing pattern (based on a conditional decision a branch is chosen), applied to Load balancing. This dimension also demonstrates the AWARD flexibility for

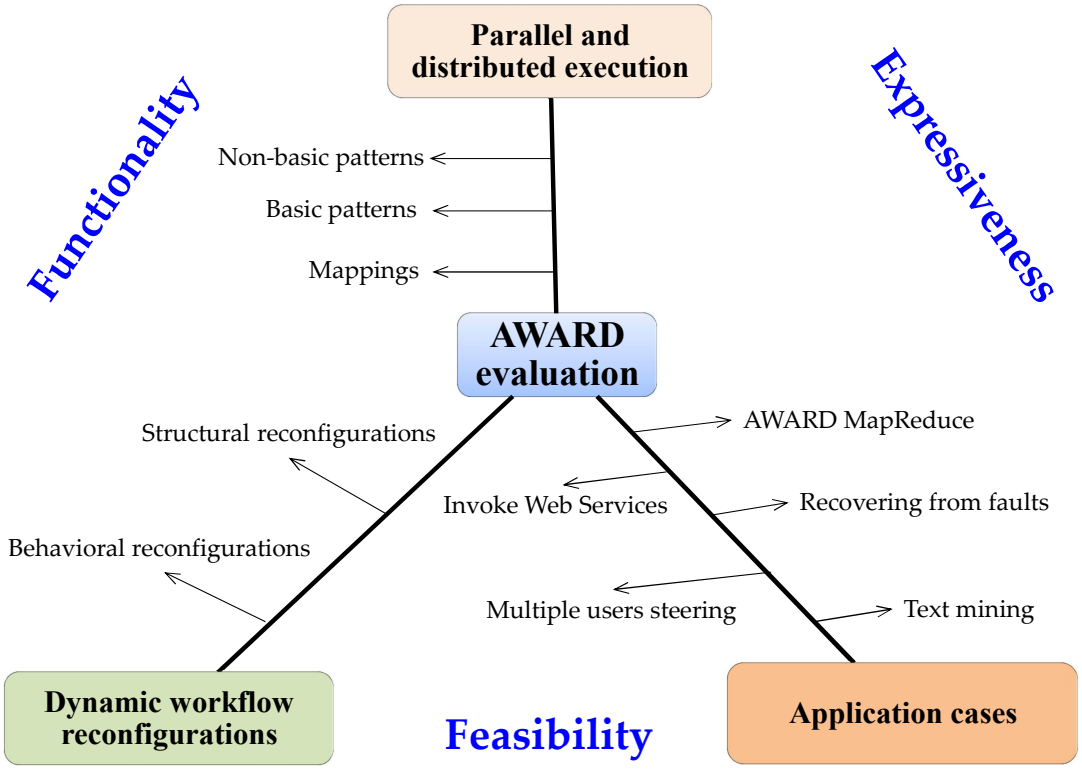


Figure 6.1: Dimensions for evaluating the AWARD framework

mapping the execution of all workflow activities on standalone computers, or allowing the execution of partitions of activities on parallel and distributed infrastructures.

✧ **Dynamic workflow reconfigurations**

Scientific experiments based on long-running workflows with multiple or possibly infinite iterations sometimes require structural and behavioral dynamic workflow reconfigurations. According to this dimension we evaluate a set of workflow scenarios that demonstrate the AWARD characteristics for supporting dynamic reconfigurations involving dynamic reconfiguration operators, available through the reusable Java library *DynamicLibrary.jar*.

✧ **Application cases**

In this dimension we evaluate the feasibility of using the AWARD framework for developing concrete workflow application cases. The following set of cases, which have been designed and implemented using AWARD, is presented: i) Using AWARD to implement programming models such as *MapReduce*, or to access Web Services; ii) Using AWARD for fault recovery by avoiding to restart entire long-running workflows; iii) Using AWARD for developing real application scenarios such as workflows with steering by multiple users, and workflows for exploring parallelism and distribution in text mining.

A set of useful workflow scenarios encompassing the above dimensions is used for evaluating the functionality, expressiveness and feasibility of the AWARD framework. Performing an analysis of the performance of the AWARD model and implementation

is out of scope of our work. However, the AWARD feasibility goal for developing real and concrete workflow applications guided the AWARD implementation to also address performance issues. Therefore on some scenarios we discuss performance indicators and the subjacent execution overheads.

6.2 Parallel and Distributed Workflow Execution

The AWARD model and the implemented architecture allow the execution of workflow activities on heterogeneous environments. The workflow activities can be mapped to standalone computers, distributed infrastructures for example, local area networks with Windows and Linux computer systems, clusters or cloud infrastructures involving large sets of virtual machines. The AWARD Space can also be mapped for execution on any virtual machine accessed from the activity virtual machines. Furthermore the AWARD model allows any data type to be used by workflow developers for specifying application-dependent tokens.

The evaluation of parallel and distributed workflow execution was performed using a set of workflow scenarios with basic and non-basic patterns and the corresponding mappings for executing the workflow activities on multiple virtual machines, allocated on local clusters and on the Amazon cloud infrastructure. Some experimental results of these scenarios have already been published in [AC13; AC14; AGC12; AGC14].

6.2.1 Mappings

AWARD workflows can be executed on standalone computers, for instance all activities and the AWARD Space can be mapped for execution on a laptop computer.

For experimenting with a distributed environment, the Amazon EC2 infrastructure [Ama15b] was used. An *Amazon EC2 Instance* is a virtual machine that can be dynamically created, with associated resources (CPUs, memory, IP addresses) and a selected operating system (in our experiments we used Linux 64 bit). *Elastic Block Storage* (EBS) volumes support virtual storage as persistent disks that can be easily attached and detached to the running EC2 Instances. Based on a volume with data files a snapshot can be created allowing the later creation of multiple volumes, all of them sharing the data files. Therefore a pool of *Amazon EC2 Instances* communicating using the TCP/IP protocol, where each instance shares (mount) data volumes, is virtually equivalent to a local network or a cluster with shared data storage. The mapping of AWARD components, the AWA workflow activities and the AWARD Space onto the Amazon EC2 infrastructure is depicted in Figure 6.2.

The number of EC2 Instances and their types in terms of computing power as well as the size of the cloud shared storage depends on a cost/benefit relation, which is defined by end users according to the application aims. As an example one EC2 Instance should

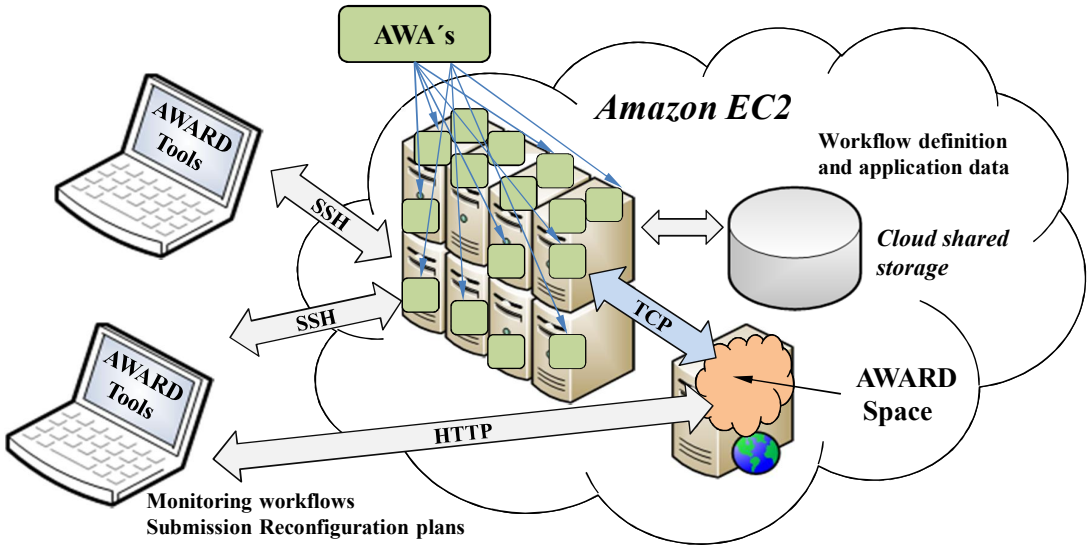


Figure 6.2: AWARD experimentation on Amazon cloud infrastructure

be enough for running simple workflows. In this case the AWARD Space and all AWA activities are launched on the same computing node.

Therefore after the workflow specification, typically the setup for performing experiments consists of the following steps:

1. Choose the EC2 Instance type and create the necessary EC2 instances using the Amazon AWS console;

2. Connect to each EC2 Instance by using any Secure Shell (SSH protocol) application for instance for copying the workflow specification file and application dependent data to the cloud shared storage;

3. Launch the AWARD Space server on a dedicated EC2 Instance accessed from the other EC2 Instances through the TCP/IP protocol. In this way the AWA workflow activities can be spread on multiple computing nodes and the tokens between input and output ports are passed through the AWARD Space using the internal Amazon communication network;

4. Launch one or more activities or even the entire workflow on EC2 Instances by using the AWARD tools described in Section 5.9;

5. Monitor the workflow execution by observing the logging information of the activities stored into the AWARD Space server, which can be accessed from anywhere outside the Amazon infrastructure by using any HTTP browser;

6. Submit dynamic reconfiguration plans by any tool developed using the available Java library (*DynamicLibrary.jar*).

6.2.2 Support for Application-dependent Tokens

Application decomposition on multiple activities that are modeled as data-flow workflows typically require the tokens to be passed between activities as application-dependent data types according to the application domain. However, some existing workflow systems lack the required flexibility regarding this issue. For instance Kepler requires tokens to be compromised or dependent on the execution engine implementation specific details.

Concerning the token expressiveness, the AWARD machine is completely decoupled from the token data types. The programmer has the freedom to specify any token datatype according to the activity *Task* implementation using the Java pattern for defining serializable classes. As an example, if a token contains a location of a database the workflow developer can program the token class as depicted in Listing 6.1 and assign the type name *DBserverInfo* to the fields *<tokenType>* of the input and output ports when specifying the workflow.

Furthermore AWARD is neutral regarding the semantics interpretation of tokens unlike other workflow systems [Shi07b] where a distinction is made between a data-flow token, when it carries data, and a control flow token, when it carries a synchronization signal. AWARD does not make any distinction between data-flow and control-flow tokens, leaving such semantics interpretation to the workflow developers according to the application requirements.

Listing 6.1: An example of an application-dependent token

```
1 public class DBserverInfo implements Serializable {
2     public String username;
3     public String password;
4     public String hostName;
5     public int tcpPort;
6     public String DatabaseNme;
7 }
```

6.2.3 Basic Workflow Patterns

Concerning the expressiveness dimension the AWARD model supports the basic workflow patterns [Aal+00a], for instance the *Sequence* pattern where an activity is enabled after the completion of another, the *Parallel split* pattern where a single activity splits into multiple activities which can be executed in parallel, and the *Join* pattern where an activity coordinates the joining of multiple parallel activities. For evaluating these basic patterns and their execution mappings on distinct environments including distributed infrastructures we developed the following two workflows, that are presented in this section:

1. A basic workflow with multiple iterations for performing arithmetic operations deterministically. For allowing comparative analyses, namely concerning the execution overheads on distinct mappings, this workflow was also developed using the

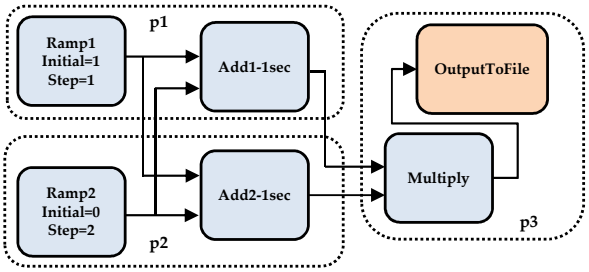
Kepler workflow system [Kep14];

2. A workflow with 25 activities that simulates a simple Montage workflow [Dee+05; Juv+10] for demonstrating the AWARD functionality for separately launching multiple partitions of workflow activities.

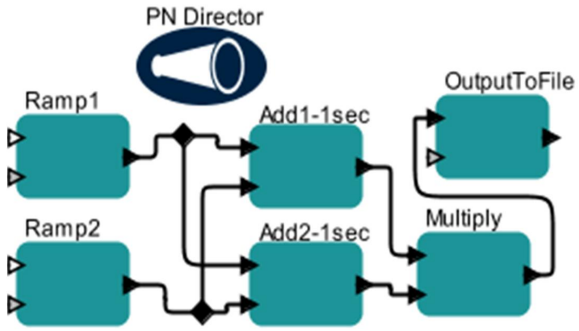
6.2.3.1 A Basic AWARD Workflow Compared with Kepler

The evaluation of basic characteristics of the AWARD model involved the execution of a similar workflow with multiple iterations using the AWARD framework and the Kepler system [Kep14] as presented in Figure 6.3, using the Kepler executable desktop application generated from the source code version 2.0.

The comparison of AWARD and Kepler was performed in several experiments including the execution of the long-running workflow and the execution of the AWARD Space server on a standalone computer or a single virtual machine on the cloud, and the execution of partitions of the workflow activities as well as the AWARD Space server on parallel and distributed infrastructures.



(a) AWARD workflow with 3 possible partitions



(b) Kepler workflow using customized *actors*

Figure 6.3: An equivalent workflow in AWARD and in Kepler

The workflow of Figure 6.3 has two *Ramp* activities for generating sequences of integer numbers until a maximum number of iterations. These numbers are added on the *Add1-1sec* and *Add2-1sec* activities and the addition results are multiplied in the *Multiply*

activity. The *OutputToFile* activity writes the results to a file whose name is defined as an activity parameter.

The workflow contains some of the basic workflow patterns. As examples, the *Sequence* pattern when the *OutputToFile* activity is only enabled after the completion of the *Multiply* activity, the *Parallel split* pattern when the *Ramp1* activity splits its output for the two add activities (*Add1-1sec* and *Add2-1sec*) executed in parallel, and the *Synchronization* pattern when an add activity, for instance *Add1-1sec* coordinates the joining of the two parallel *Ramp* activities.

Both AWARD and Kepler support parallelism in the execution of the workflow patterns. In Kepler all activities are executed by a single operating system process and the parallelism is enforced by the PN Director [God+09] where each workflow activity (*actor*) is executed in a Java thread, and all *actors* are executed concurrently communicating through memory buffers. Therefore in Kepler the control for executing the workflow activities is centralized. In fact, to the best of our knowledge and according to the available publications and the available software distribution, Kepler does not allow a decentralized control for executing workflow activities in distinct operating system processes and much less on distinct computing nodes. In AWARD the parallelism is much more flexible due to the autonomic characteristics of each workflow activity. In fact in AWARD activities can be separately launched in the same computing node as distinct operating system processes or even distinct processes on distinct computing nodes.

For simulating long elapsed execution times, in this experiment the AWARD *Add1* and *Add2* activities have a *Task* with a sleep time of 1 second, this time being defined as a *Task* parameter as presented in Listing 6.2.

For achieving the same behavior in Kepler we customized the corresponding *actors*. As an example, Listing 6.3 presents the Kepler *actor* to add two numbers with the same sleep time of 1 second and to manage the workflow iterations.

In a similar way as the AWARD capability for logging execution times on the AWARD Space that can be retrieved by the *DumpExecTimes.jar* tool, the Kepler *actor* reports the execution time per iteration into a log file located in an operating system temporary directory.

As a first evaluation comment the simplicity of developing AWARD *Tasks* (Listing 6.2) should be noted when compared to Kepler (Listing 6.3 on page 202).

In Kepler the programmer needs to deal with low-level details related to the execution engine, for instance for creating the input and output ports and for getting their tokens.

In AWARD these actions are transparently performed by the *Autonomic Controller* based on the workflow specification.

The experiments were executed on the Amazon EC2 infrastructure as depicted in Figure 6.2 on page 197 and used an *EC2 Instance* type with 4 virtual CPU (2 virtual cores with 2 EC2 Compute Units each), 7.5GB memory, 8 GB root device, and a 8 GB shared volume running the Linux operating system.

Listing 6.2: AWARD *Task* to add two numbers with a delay of 1 second

```
1  package awardutiltasklibrary;
2
3  import awardtaskinterface.IGenericTask;
4  import awardlog.logging;
5
6  /** @author lassuncao */
7  public class TaskIntegerAdd implements IGenericTask {
8
9     public Object[] EntryPoint(Object[] args, Object[] params) throws Exception {
10       Object[] Results = new Object[1];
11       Integer res = null;
12       Integer del = null;
13       try {
14          Integer x = (Integer) args[0];
15          Integer y = (Integer) args[1];
16          res = new Integer(x.intValue() + y.intValue());
17          Results[0] = res;
18          if (params != null) {
19             del = new Integer((String) params[0]);
20             Thread.sleep(del.intValue());
21          }
22       } catch (InterruptedException ex) {
23          AwardLog.logging(3, "Exception in Task TaskIntegerAdd");
24        } catch (Exception uex) {
25          throw uex;
26       }
27       return Results;
28    }
29 }
```

For executing the Kepler workflow we used a single *EC2 Instance* because Kepler does not support the execution of workflow partitions.

For executing AWARD workflow we considered four cases:

- **Case 1**: We ran the workflow using a single *EC2 Instance*, including the AWARD Space server for allowing a fair comparison with the Kepler execution;
- **Case 2**: We used two EC2 instances, one to host the AWARD Space server and the other to execute the workflow activities;
- **Case 3**: We used four EC2 instances, one to host the AWARD Space server and three EC2 instances to distribute the activities according to the p1, p2 and p3 workflow partitions, as shown in Figure 6.3(a). The mappings of the workflow activities and the AWARD Space to four EC2 Instances are illustrated in Figure 6.4;
- **Case 4**: We ran the workflow using a single *EC2 Instance*, including the AWARD Space server, using a more powerful and expensive *EC2 Instance* with 8 virtual CPU and 60 GB memory with high network performance.

Listing 6.3: Kepler *actor* for adding two numbers with a delay of 1 second

```
package actorkepler;
import java.io.*;
import ptolemy.actor.*;
import ptolemy.data.Token;
import ptolemy.kernel.*;
/** @author lassuncao */
public class AddDelay1Second extends TypedAtomicActor {
    public AddDelay1Second(CompositeEntity container, String name)
                        throws IllegalActionException, NameDuplicationException {
        super(container, name);
        input1 = new TypedIOPort(this, "input1", true, false);
        input2 = new TypedIOPort(this, "input2", true, false);
        output = new TypedIOPort(this, "output", false, true);
        output.setTypeAtLeast(input1); output.setTypeAtLeast(input2);
        curIter = 0;
        filetimes = File.separator + "tmp" + File.separator + this.getName() + ".txt";
    }
    private String filetimes; /* To store the iterations elapsed execution time */
    /* Input and output ports for tokens */
    public TypedIOPort input1;
    public TypedIOPort input2;
    public TypedIOPort output;
    private int curIter; /* Current iteration: Counts the number of actor fires */

    public Object clone(Workspace workspace) throws CloneNotSupportedException {
        AddDelay1Second newObject = (AddDelay1Second) super.clone(workspace);
        newObject.output.setTypeAtLeast(newObject.input1);
        newObject.output.setTypeAtLeast(newObject.input2);
        return newObject;
    }
    /* Add tokens on the input ports. Fired by each iteration */
    public void fire() throws IllegalActionException {
        super.fire();
        Token sum = null;
        curIter++;
        FileWriter fstream = null;
        try {
            fstream = new FileWriter(filetimes, true);
            BufferedWriter out = new BufferedWriter(fstream);
            long ts = System.currentTimeMillis();
            out.write(this.getName() + ", " + curIter + ", fire, " + ts + "\n");
            if ((input1.getWidth() > 0) && (input2.getWidth() > 0) &&
                 input1.hasToken(0) && input2.hasToken(0)) {
                   sum = input1.get(0);
                   sum = sum.add(input2.get(0));
            } else throw new IllegalActionException("Illegal Firing");
            out.flush();
            fstream.close();
            Thread.sleep(1 * 1000); /* Sleep 1 second */
            if (sum != null) {
               output.send(0, sum);
            }
        } catch (InterruptedException ex) {
            throw new IllegalActionException("AddDelay actor: Sleep exception");
        } catch (IOException ex) {
            throw new IllegalActionException("AddDelay actor: IO exception");
        }
    }
}
```

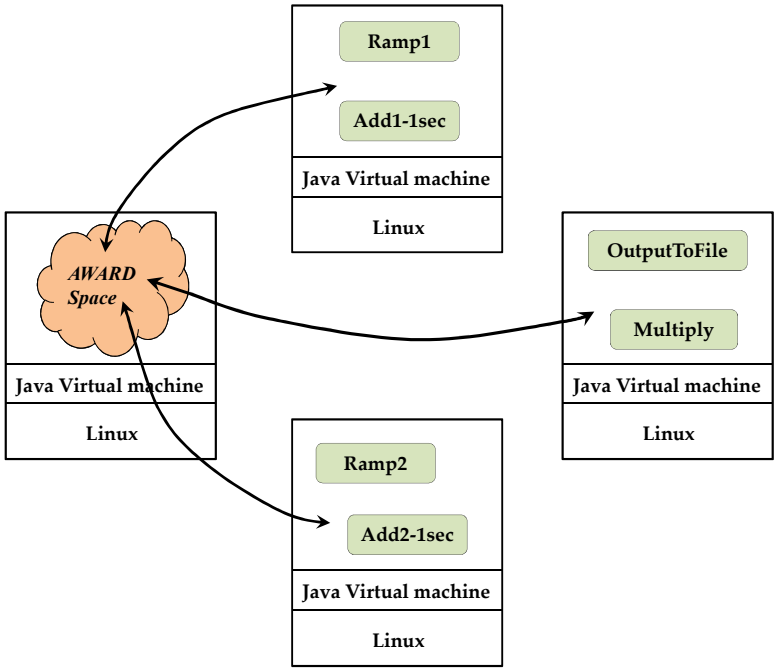


Figure 6.4: Mapping workflow partitions to multiple EC2 instances (Case 3)

The average execution time per iteration is shown in Figure 6.5 for executions with 1, 10, 50, 100, 500 and 1000 iterations. These results confirm that the current implementation of AWARD has higher overheads compared to the used implementation of Kepler, version 2.0. This is due to the fact that the Kepler implementation uses a centralized execution engine with a dedicated thread for each *actor* inside a single monolithic process and the *actors* exchange data over links implemented in memory, whereas the AWARD implementation uses an independent process for each AWA activity and exchanges tokens through the AWARD Space server using TCP/IP connections. Then for a small number of iterations the AWARD execution time is mostly penalized by the inherent overheads originated by the execution of multiple decentralized processes and the overheads associated to the TCP/IP communication with the AWARD Space server.

However, as shown in Figure 6.5, for long-running workflows with thousands of iterations, AWARD exhibits only small overheads when compared to Kepler, allowing us to conclude that AWARD becomes adequate to execute such class of workflows. For example in case 4 (single powerful *EC2 instance*) for 1000 iterations the AWARD average iteration execution time is only 4 milliseconds higher when compared with the Kepler case. For case 3 (4 EC2 instances for executing the p1, p2, and p3 workflow partitions) the execution time is smaller compared to case 1 and 2, for distributed execution on multiple EC2 instances, even for lower numbers of iterations. However, the execution time reduction is not so strong when the iterations increase up to the thousands. This is explained by the cost due to the tuple matching and tuple persistence operations when the size of the tuple space grows inside the AWARD Space server.

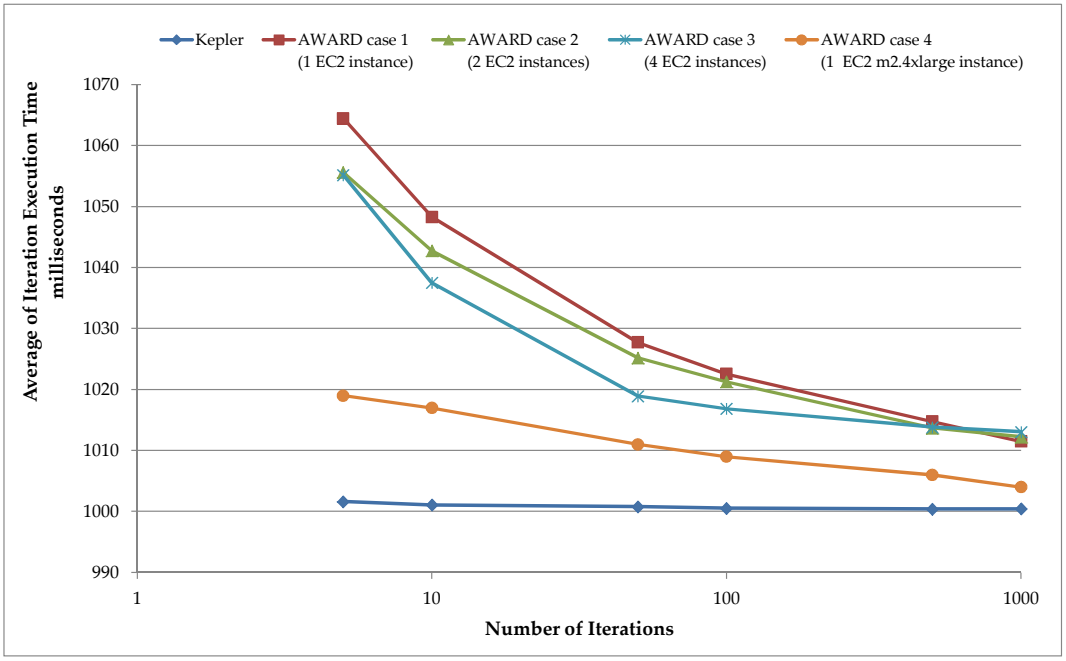


Figure 6.5: Average of iteration execution time

6.2.3.2 A Montage Workflow Simulation

For demonstrating the AWARD functionality and feasibility to support parallel and distributed execution of workflows with a large number of activities we developed a workflow with 25 activities that simulates a simple Montage workflow [Dee+05; Juv+10].

The AWARD flexibility allows workflow activities to be separately launched by different users without a centralized control. Furthermore the AWARD flexibility supports easily changing the workflow activity mappings to distinct virtual machines without any modification to the workflow specification.

The software toolkit Montage [Mon15] which provides tools for processing astronomical image mosaics is used on Montage workflows involving thousands of activities for modeling experiments to construct mosaics centered on celestial objects [Dee+05; Juv+10]. However, without considering the activities used to transfer the image data files a Montage workflow, as presented in [Dee+05], has a structure with seven levels as depicted in Figure 6.6.

As presented in [Dee+05], the workflow development assumed that for levels with multiple activities the activity *Tasks* at each level are equal and have the reference elapsed execution time that are indicated in the right side of Figure 6.6, respectively for each level from 1 to 7.

Therefore considering the sequential dependencies between the levels and the parallelism among the activities within each level allow considering the sum of all level execution times, that is, 147.7 seconds as a reference time for the total workflow execution time. This value was used in the experiment as a reference time to evaluate the

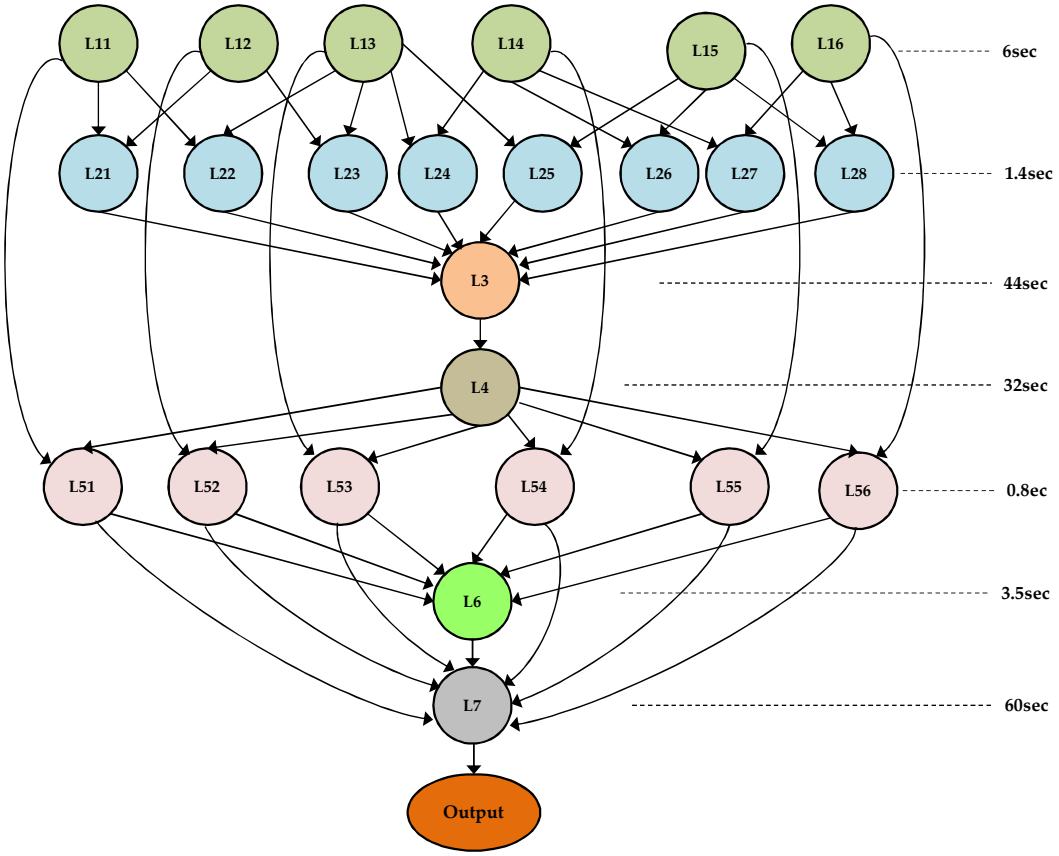


Figure 6.6: The structure of a Montage workflow, [Dee+05]

performance (total execution time) which was obtained by AWARD workflows in different execution scenarios.

For evaluating performance and overhead indicators of the AWARD framework implementation we developed the Montage workflow with 25 activities presented in Figure 6.6 where the *Output* activity is used to visualize the final workflow result, as presented in Figure 6.7. In order to simulate the behavior of the Montage workflow each AWARD activity *Task* imposes a delay of a constant execution time equal to the indicated on the right side of Figure 6.6.

Each activity produces tokens with the "$ActivityName(TokenInput_1,...,TokenInput_N)$" string type pattern on its N outputs, where "$TokenInput_1,...,TokenInput_N$" is a list of tokens received on the activity input ports or a list of activity *Parameters*.

As an example the L_{11} activity that has two *Parameters* (image1 and header) produces, on its three outputs, the "$L_{11}(image1, header)$" string and similarly the L_{12} activity produces the "$L_{12}(image2, header)$" string.

Therefore the L_{21} activity produces the "$L_{21}(L_{11}(image1, header), L_{12}(image2, header))$" string on its unique output. When the workflow execution finishes, the *Output* activity shows the workflow result as illustrated in Figure 6.7.

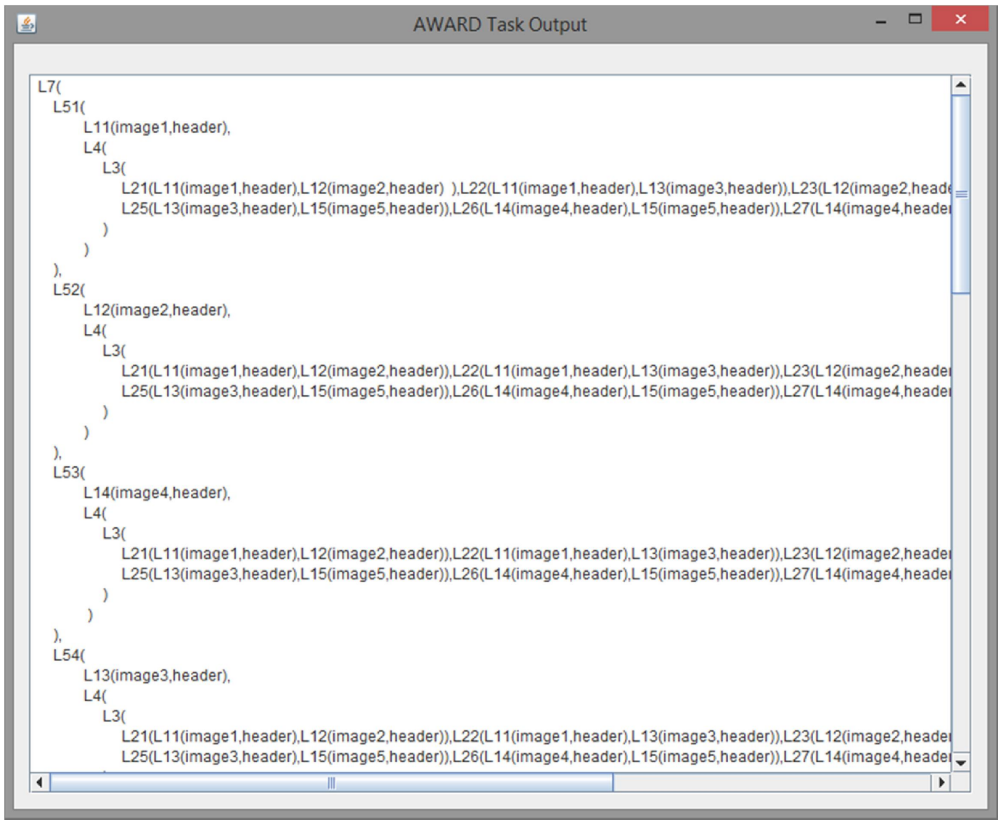


Figure 6.7: The activity Output shows the workflow result

This workflow enabled multiple experimental scenarios for evaluating the performance and overheads associated to the execution of the AWARD workflows as well as the AWARD flexibility for executing workflows on distributed infrastructures where multiple users can execute a large workflow interactively by executing their specific partitions of activities.

✧ **Scenario 1: Executing the Montage workflow and AWARD Space server on a standalone computer**

According to this first scenario we launched the workflow on a laptop computer (Windows 8, 64 bit, 8GB RAM, i7-368-Dual core, 2.1GHz) using the AWARD tool presented in Section 5.9.2 on page 190. This means that the AWARD Space server and the 25 workflow activities ran in the same computing node.

To calculate the elapsed workflow execution time we used the AWARD functionality for supporting logs related to the steps of the *Autonomic Controller* of each activity, namely the execution times retrieved by the *DumpExecTimes.jar* AWARD tool. Therefore we considered the difference between the BO (Before Output) time of the L7 activity and the BI (Before Input) lower time of the first level activities {L11,...,L16}.

The average of the elapsed execution time achieved from various executions was 160.4 seconds which allows to calculate an overhead of 12,7 seconds, as the difference between 160.4 and the Montage workflow reference execution time of all activities (147.7 seconds).

This overhead of 8.6% includes the operating system overheads for scheduling all 25 executors of AWA activities (*AwaExecutor*) as well as a negligible overhead of 157 milliseconds caused by locally accessing the AWARD Space server for issuing and retrieving the 51 string tokens passed between activities.

This scenario allows concluding that the overhead for executing workflows with 25 workflow activities on a standalone computer is acceptable. It is important to note that other operating system processes (Windows 8 services) were running on the laptop computer during the experiment.

✧ **Executing the Montage workflow and AWARD Space server on the Amazon cloud**

In this scenario we used the Amazon EC2 infrastructure as described in Section 6.2.1 for experimenting with two approaches for launching the workflow of Figure 6.6.

The first approach consists of launching all activities and the AWARD Space server in the same *EC2 Instance*.

The second approach consists of launching different workflow partitions by different users on multiple EC2 instances for exploring distributed executions. Considering the heuristics that some activities can only run after the completion of an upstream group of activities, three possible partitions (p1, p2, p3) can be launched by 3 users in 3 different EC2 instances as depicted on Figure 6.8.

These partitions are explicitly defined by the user according to a decomposition strategy depending on the application knowledge. Another possible heuristics could be based on strategies for optimizing the utilization of computational resources by only launching the workflow partitions that can be run in the near future. For instance, [Dee+05] evaluates the benefits of task clustering for optimizing the scheduling of workflow tasks to the available resources using the Pegasus workflow system.

The type of EC2 Instances used in this 2nd scenario was m2.4xlarge (8 virtual CPU and 60 GB memory).

In the first approach, when launching the entire workflow including the AWARD Space server in the same EC2 Instance, the elapsed execution time was 153.4 seconds which decreases the overhead relative to Montage reference time, to 5.7 seconds (153.4-147.7), that is 3,9% instead of the value of 8.6% previously obtained on the laptop standalone computer. The overhead for locally accessing the AWARD Space server for issuing and retrieving 51 string tokens passed between activities was 168 milliseconds, similar to the laptop computer overhead.

In the second approach we used four EC2 Instances (8 virtual CPU and 60 GB memory), one dedicated *EC2 Instance* to run the AWARD Space server and three EC2 instances to run the workflow partitions (p1, p2, p3) as depicted in Figure 6.8. The elapsed execution time was 154.2 seconds which is slightly above (800 milliseconds) the time obtained when using a single *EC2 Instance*. This can be partially justified by the greater overhead of 234 milliseconds related to Amazon network communication between EC2 instances to access the AWARD Space server.

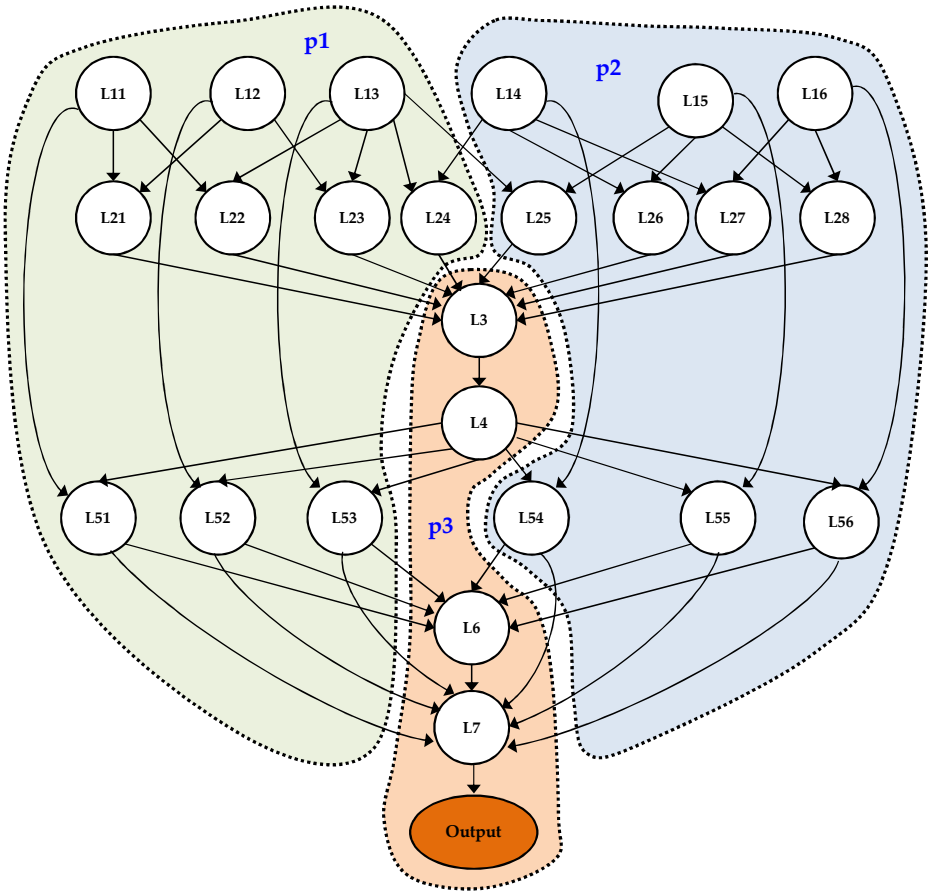


Figure 6.8: Workflow partitions (p1, p2, p3) to be separately launched

Although the distributed execution of the three partitions (p1, p2, p3) does not benefit the execution performance this scenario allows concluding in favor of the AWARD flexibility to execute workflows with a large number of activities that can be separately launched by distinct users on distributed computing nodes.

Both scenarios allow concluding that the communication overheads for accessing the AWARD Space server do not have a significant weight in the workflow elapsed execution time.

6.2.4 Non-basic Workflow Patterns

The approach of modeling application decomposition using workflows requires flexibility [Gil+07; Shi07b] to develop workflow patterns not supported in most of the existing and widely used workflow systems. Also in our preliminary work [AGC09] we identified the need for a non-basic pattern of a feedback loop.

Therefore one important motivation for developing the AWARD model was the flexibility needed to support some non-basic but useful workflow patterns.

In the following subsections we present examples of workflows to demonstrate some non-basic patterns, such as multi-merge, feedback loop and load balancing patterns.

6.2.4.1 Multi-merge Pattern

The multi-merge pattern [Aal+00b] is used to model a point in a workflow where multiple branches converge without synchronization. If multiple branches have tokens, possibly produced concurrently, the merge activity is started for any activation of any incoming branch.

The AWARD model supports this pattern as illustrated in Figure 6.9 by using a multi-link join (Definition 3.11 on page 62) to an input port named *Ci1* configured according to the *Any* order mode corresponding to a non-deterministic merge.

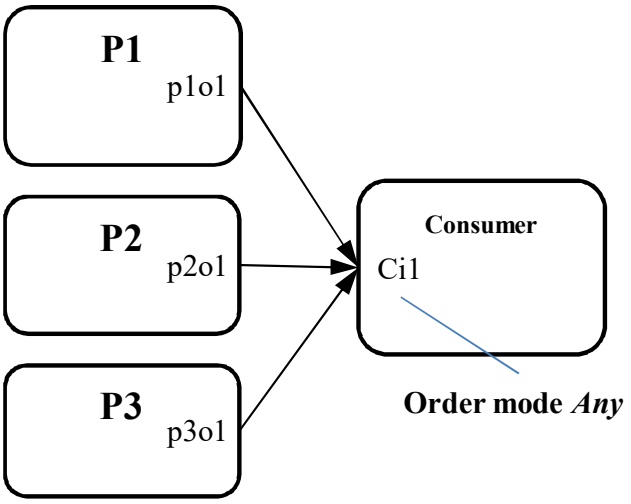


Figure 6.9: A Multi merge pattern (Producer/Consumer)

For evaluating this pattern the workflow of Figure 6.9 simulates the producer/consumer pattern. The *P1*, *P2* and *P3* activities are executed in parallel and used for producing tokens on their *p1o1*, *p2o1* and *p3o1* output ports. These tokens are consumed in a non-deterministic order by the *Consumer* activity that has an input port (*Ci*1) configured to the *Any* order mode.

The relevant details of the workflow specification are presented in Listing 6.4 where it is relevant to note that the *Consumer* activity specifies a maximum number of iterations for overriding the specification of *MaxIterations* as the global number of iterations. This is due to the fact that if each producer activity has 20 iterations, then the *Consumer* activity must execute 60 iterations for consuming the total of 60 tokens produced by *P1*, *P2* and *P3*.

The workflow execution result is shown in Figure 6.10 where the non-deterministic behavior of the *Ci1* input port is clearly visible. For instance the first token produced by *P3* at iteration 0 is only consumed at the 20^{th} iteration of the *Consumer* activity.

Listing 6.4: Specification details of the Multi merge workflow (Figure 6.9)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AwardWorkflow>
    <workflowName>workflow Producer/Consumer</workflowName>
    <MaxIterations>20</MaxIterations>
    <!-- . . . -->
    <Awa> <!-- Activity Producer P1 -->
        <!-- . . . -->
        <name>P1</name>
        <output>
           <name>p1o1</name>
           <state>Enable</state>
           <tokenType>java.lang.String</tokenType>
           <modeToken>Single</modeToken>
           <sendTo>Ci1</sendTo>
        </output>
        <Task>
            <parameters> <par>P1</par> </parameters>
            <SoftwareComponent>
               <taskImplementationType>tasks.TaskProducer</taskImplementationType>
               <mappingResults>
                  <result> <idxRes>0</idxRes> <outName>p1o1</outName> </result>
               </mappingResults>
            </SoftwareComponent>
        </Task>
    </Awa>
    <!-- . . . P2 and P3 AWA activities are similar to P1 -->
    <Awa> <!-- Activity Consumer -->
        <ControlUnit>
           <MaxIterations>60</MaxIterations>
           <InitialState>Idle</InitialState>
           <RulesFilePath>TaskBasicRules.clp</RulesFilePath>
        </ControlUnit>
        <name>Consumer</name>
        <input>
           <name>Ci1</name>
           <state>Enable</state>
           <tokenType>java.lang.String</tokenType>
           <orderMode>Any</orderMode>
        </input>
        <Task>
            <parameters> <par>C</par> </parameters>
            <SoftwareComponent>
               <mappingArgs>
                  <arg>
                     <idxArg>0</idxArg>
                     <inName>Ci1</inName>
                  </arg>
               </mappingArgs>
               <taskImplementationType>tasks.TaskConsumer</taskImplementationType>
            </SoftwareComponent>
        </Task>
    </Awa>
</AwardWorkflow>
```

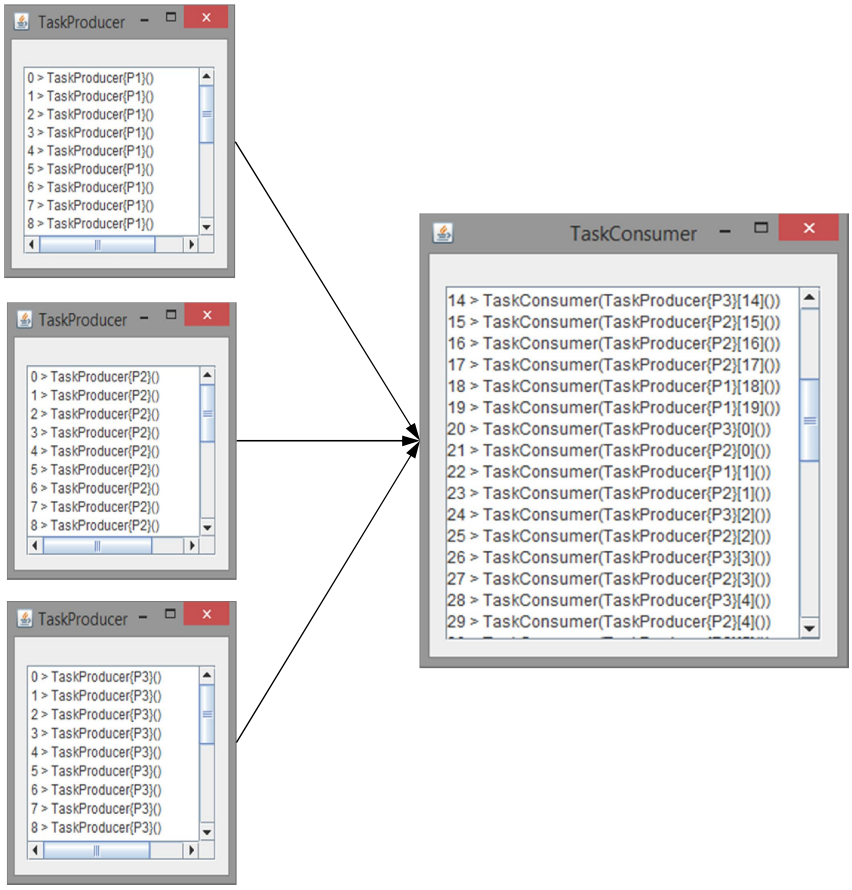


Figure 6.10: The multi-merge workflow execution

6.2.4.2 Feedback Loop Pattern

When a workflow has multiple iterations it can be useful that an activity *Task* at iteration K knows data produced in the previous iteration $(K-1)$ by any downstream activity. The *Feedback loop* pattern consists of workflow activities that have input ports connected to output ports of activities executed in a downstream sequence. For the evaluation of the non-basic *Feedback loop* pattern supported by the AWARD model we used the workflow depicted in Figure 6.11.

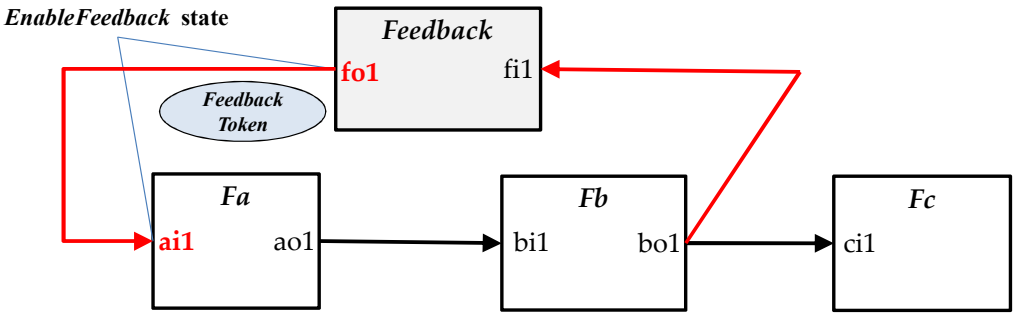


Figure 6.11: Workflow with a feedback loop pattern

The workflow has three activities, *Fa*, *Fb*, and *Fc* for processing a sequence of data tokens as a pipeline during multiple iterations. However, on iteration *K* the *Fa* activity receives, on its ai1 input port, feedback tokens that the *Feedback* activity emitted according to the data token produced, in the $(K-1)$ iteration, by the *Fb* activity, on its *bo1* output port connected by a multi-link split to the ports *fi1* of the *Feedback* activity and *ci1* of the *Fc* activity.

For supporting this pattern the workflow developer specifies the *EnableFeedback* state (Definition 3.15 on page 64) for defining the behavior of the *fo1* output port of the *Feedback* activity as well as for the *ai1* input port of the *Fa* activity. This means that at iteration 0 the *ai1* input port of the *Fa* activity is disabled and will be automatically enabled on the 1st iteration at that activity and the *fo1* output port of the *Feedback* activity always produces tokens marked with the current iteration number of the *Feedback* activity plus 1.

The relevant XML fragments of the workflow specification involved in the feedback loop, in particular the specification of the *bo1* and *fo1* output ports and the *ai1* input port, are presented in Listing 6.5.

For demonstrating an operational example and visualize what happened on each activity we developed activity *Tasks* for all activities with a graphical user interface as shown in Figure 6.12. For each iteration these *Tasks* produce strings to be sent through output ports with the following format "*ActivityName(<string received on input port>)*". Thus on each iteration we visualize the feedback effect. For instance on the 1st iteration the *Fa* activity received the "*Feedback[it:0](Fb[it:0](Fa[it:0]()))*" string token which can be interpreted as - *the token was issued by the Feedback activity at iteration 0 that had received a token from the Fb activity at iteration 0 which in turn had received a token from the Fa activity at iteration 0*.

Although the workflow structure used in this example shows the *Feedback* activity between the *Fb* activity and the *Fa* activity, many other structures can be used, for instance, the feedback loop can be placed between the last *Fc* activity and any other workflow activities for carrying intermediate results. This is depicted in Figure 6.13 where the intermediate result of each iteration produced on the *co1* output port gives feedback information to the *Fb* activity on its *bif* input port.

This example assumes that the workflow developer is able to specify the feedback loop at design time. However, in real applications the need for getting feedback data in order to optimize a particular activity *Task* can often be only recognized during workflow execution by observation of some application dependent intermediate results. In Section 4.4.4 the scenario of Figure 6.13 is demonstrated where the *Feedback loop* pattern is dynamically introduced by submitting a dynamic reconfiguration plan for changing the structure of the long-running workflow originally composed only of the *Fa*, *Fb*, and *Fc* activities.

Listing 6.5: Relevant details of the workflow specification involved in a feedback loop

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AwardWorkflow>
    <workflowName>workflow Feedback Loop</workflowName>
    <MaxIterations>5</MaxIterations> <!-- Number of Iterations -->
    <!-- . . .-->
    <Awa> <!-- Activity Fa -->
        <!-- . . .-->
        <name>Fa</name>
        <!-- . . .-->
        <input>
          <name>ai1</name>
          <state>EnableFeedback</state>
          <tokenType>java.lang.String</tokenType>
          <orderMode>Iteration</orderMode>
        </input>
        <!-- . . .-->
    </Awa>

    <Awa> <!-- Activity Fb -->
        <!-- . . .-->
        <name>Fb</name>
        <!-- . . .-->
        <output>
          <name>bo1</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>Replicate</modeToken>
          <sendTo>ci1</sendTo>
          <sendTo>fi1</sendTo>
        </output>
        <!-- . . .-->
    </Awa>

    <Awa> <!-- Activity Feedback -->
        <!-- . . .-->
        <name>Feedback</name>
        <!-- . . .-->
        <output>
          <name>fo1</name>
          <state>EnableFeedback</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>Single</modeToken>
          <sendTo>ai1</sendTo>
        </output>
        <!-- . . .-->
    </Awa>
</AwardWorkflow>
```

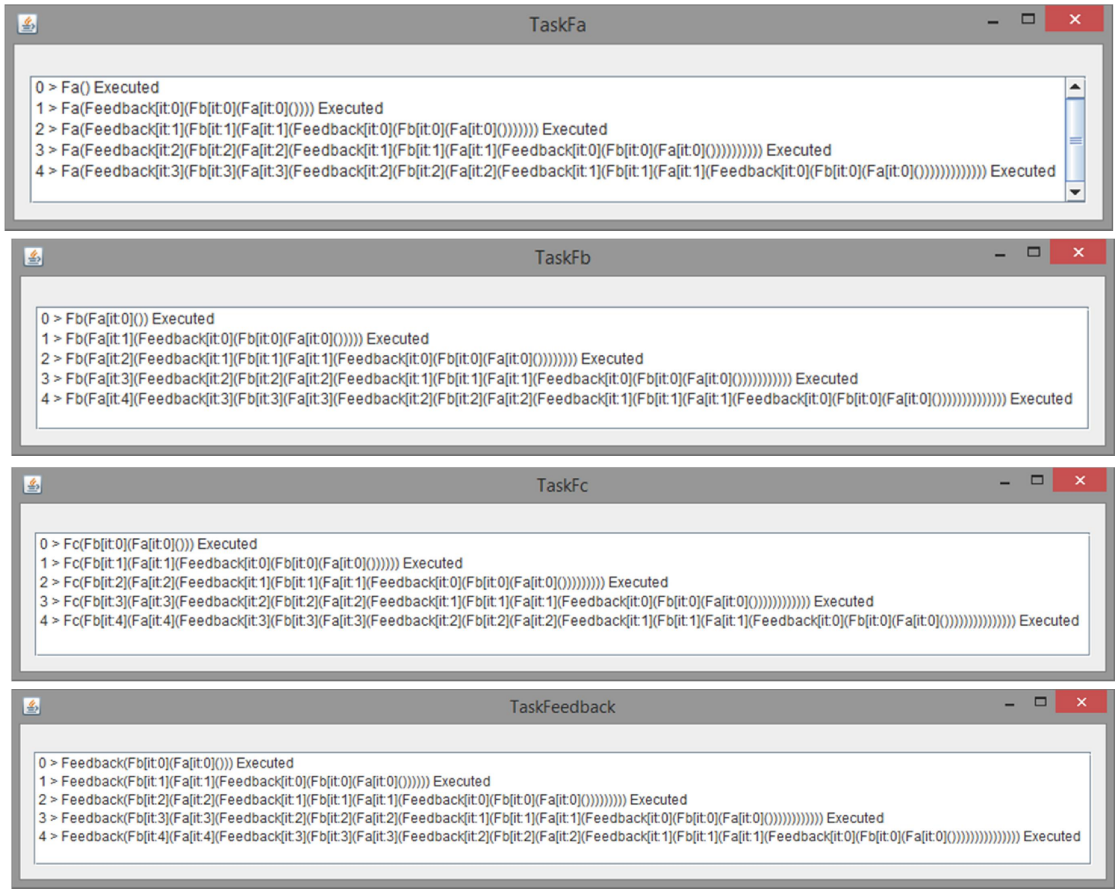


Figure 6.12: The execution of *Fa*, *Fb*, *Fc* and *Feedback* activities for 5 iterations

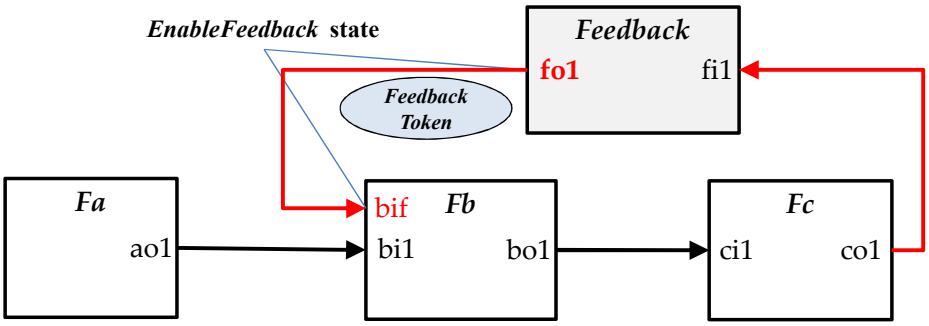


Figure 6.13: Other possible workflow structure with a feedback loop

6.2.4.3 Load Balancing Pattern

The load balancing pattern supported by the AWARD model is very important for improving execution performance of activities with heavy load characteristics. Although an application can be decomposed into multiple workflow activities, the granularity of some activities can be constrained by algorithm or data restrictions. Thus, if an overloaded workflow activity exhibits a high execution time then other downstream activities are penalized by having to wait for tokens produced by the overloaded activity and as a result

the total workflow execution time tends to increase.

According to the application requirements a possible solution to overcome this issue is scaling out the overloaded activity with a number of replicas for load distribution. This requires that the upstream activity has the capability for distributing tokens among all the replicas of the scaled-out activity.

For demonstrating the load balancing pattern we considered the simple pipeline workflow presented in Figure 6.14 where the *Delay* activity is overloaded, thus having a significant elapsed execution time. Therefore despite the high rates of the *Ramp* and *Output* activities for processing tokens, the *Output* activity is delayed because it depends on the execution pace of the *Delay* activity.

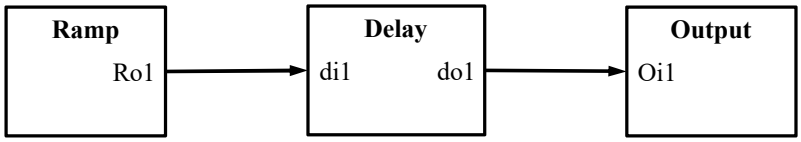


Figure 6.14: Pipeline workflow with an overloaded activity

This issue can be overcome by using the load balancing pattern supported by the AWARD model allowing tokens produced by the *Ramp* activity to be distributed to multiple parallel replicas of the *Delay* activity.

The workflow depicted in Figure 6.15 demonstrates the load balancing pattern applied to 1 up to 6 replicas of the *Delay* activity.

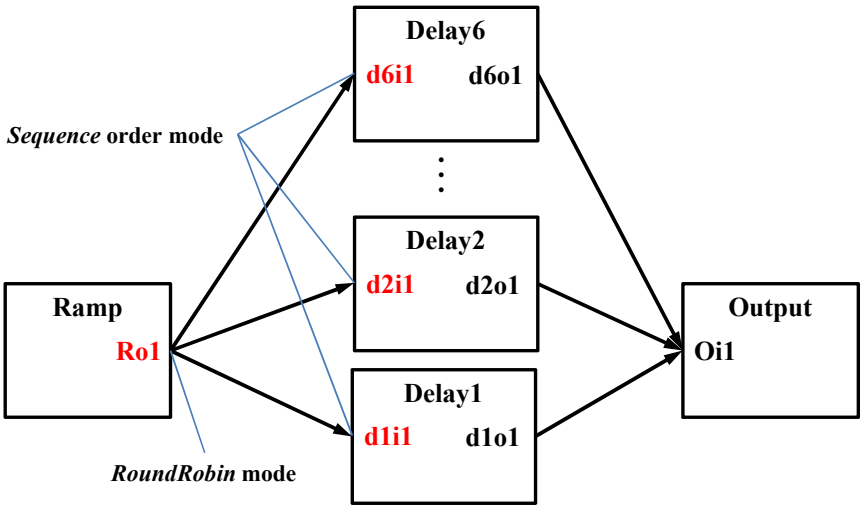


Figure 6.15: Workflow with a load balancing pattern

The *Ramp* activity produces string tokens on its *Ro1* output port configured in the *RoundRobin* mode (Definition 3.10 on page 61) for sending tokens in round-robin fashion to all destinations connected to the *Ro1* output port. Each token carries a string with the system time in milliseconds and is marked with a sequential number for each destination.

The destination input ports are configured in the *Sequence* order mode (Definition 3.4 on page 58) for receiving tokens in their sequence number order.

The *Output* activity receives tokens on its *Oi1* input port (configured in the *Iteration* input mode) from multiple sources without knowledge on how many token emitters exist and shows the result strings for each iteration. This is presented in Figure 6.16 for a workflow specification with six replicas of the *Delay* activity. As an example, note that the token received in the *Output* activity in the 7^{th} iteration (6> in Figure 6.16) and produced by the *Ramp* activity in the 7^{th} iteration was processed by the *Delay*1 activity on its 2^{nd} iteration.

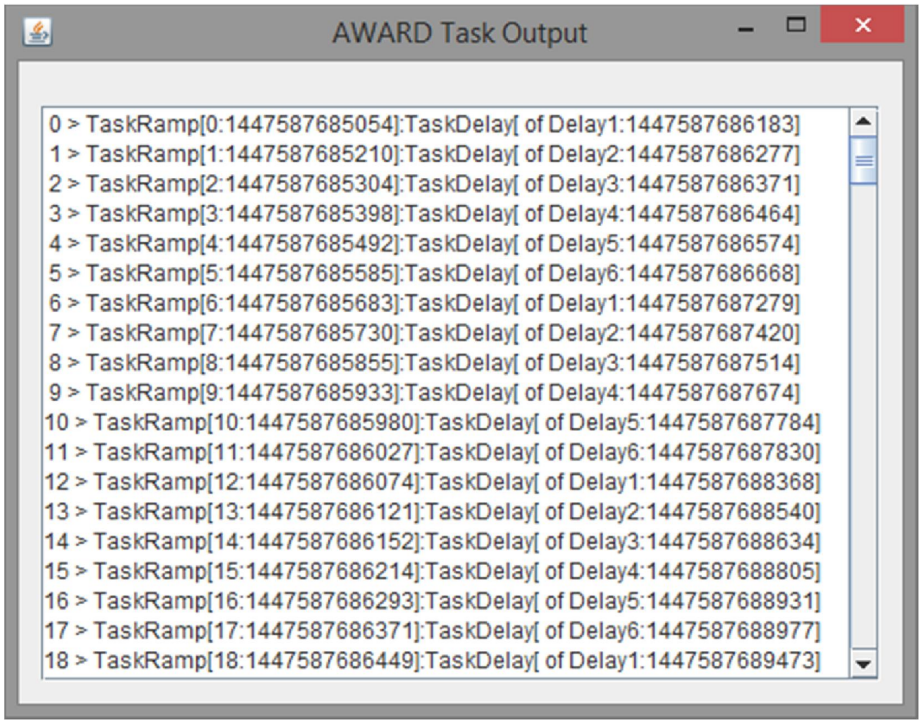


Figure 6.16: The *Task* of *Output* activity shows the iterations result

For evaluating the benefits of using this pattern we specified four versions of the workflow for executing 200 iterations that have respectively 1, 2, 4 and 6 replicas of the *Delay* activity. The *Task* of the *Delay* activity sleeps during 1 second and then the *Delay* activity sends a token with the system time to the *Output* activity. Therefore when a single *Delay* activity is used the total workflow execution time is always greater than 200 seconds.

Listing 6.6 presents the relevant details of the workflow specification when six replicas of the *Delay* activity are used. In particular, Listing 6.6 shows the specification of the *RoundRobin* mode of the output port of the *Ramp* activity and the *Sequence* order mode of the input port of the first replica, named *Delay1*.

As illustrated in Figure 6.17 the total workflow execution time with a single *Delay* activity is 215 seconds where 200 seconds are spent by the *Delay* activity *Task*, as sleep

Listing 6.6: Relevant details of the workflow specification in the load balancing pattern

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AwardWorkflow>
    <workflowName>Load Balancer</workflowName>
    <MaxIterations>200</MaxIterations> <!-- Number of workflow iterations -->
    <!-- . . . -->
    <Awa> <!-- Activity Ramp -->
        <!-- . . . -->
        <name>Ramp</name>
        <output>
          <name>Ro1</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>RoundRobin</modeToken>
          <sendTo>d1i1</sendTo>
          <sendTo>d2i1</sendTo>
          <sendTo>d3i1</sendTo>
          <sendTo>d4i1</sendTo>
          <sendTo>d5i1</sendTo>
          <sendTo>d6i1</sendTo>
        </output>
        <!-- . . . -->
    </Awa>
    <Awa> <!-- Activity LB Delay1 -->
        <!-- . . . -->
        <name>Delay1</name>
        <input>
          <name>d1i1</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <orderMode>Sequence</orderMode>
        </input>
        <output>
          <name>d1o1</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>Single</modeToken>
          <sendTo>Oi1</sendTo>
        </output>
        <!-- . . . -->
    </Awa>
    <Awa> <!-- Activity LB Delay2 -->
        <!-- . . . -->
        <name>Delay2</name>
        <!-- . . . -->
</AwardWorkflow>
```

time during 200 iterations. However, the total execution time decreases significantly when multiple replicas of the *Delay* activity are used for load balancing purposes. For instance for six replicas of the *Delay* activity the total execution time decreases to 37 seconds. Considering a speedup definition, as $Speedup = \frac{ExecTime_{replicas\#1}}{ExecTime_{replicas\#6}} = \frac{215}{37} = 5.8$ we achieved a quasi-linear speedup.

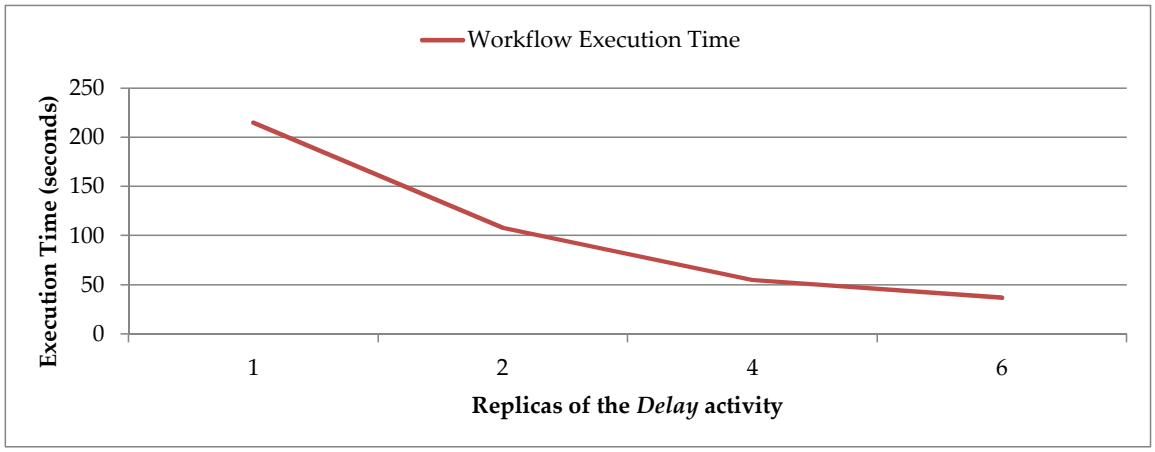


Figure 6.17: Execution times with load balancing pattern in the *Delay* activity

6.2.5 Section Conclusions

The evaluation examples used for discussing the AWARD model concerning the support of basic and non-basic workflow patterns demonstrated the AWARD flexibility and expressiveness for developing and executing workflows.

The following characteristics are emphasized:

- Functionality and expressiveness for using basic and non-basic workflow patterns;
- Feasibility for executing workflows on standalone computers and distributed computers, namely on the Amazon cloud.

The evaluation examples also have shown that the workflow activities or the partitions of workflow activities can be separately launched on different computing nodes.

The overheads involved particularly for accessing the AWARD Space server are acceptable without a significant impact upon the workflow elapsed execution time.

6.3 Evaluating the Support for Dynamic Reconfigurations

The flexibility for supporting dynamic workflow reconfigurations is a distinctive characteristic of the AWARD model. In fact, the AWARD functionality and expressiveness for supporting dynamic, adaptive, and user-steered reconfigurable workflows enable distributed and collaborative scientific experiments. Furthermore, the AWARD flexibility

allows a continuous improvement, adaptation and recovery from failures during workflow execution by supporting dynamic changes, for instance, in the activity *Tasks* in order to increase the workflow performance.

The transparency provided by the API of the *DynamicLibrary.jar* contributes to simplifying the programming of reconfiguration plan scripts as simple application-dependent tools. In the following we discuss a set of experimental scenarios for evaluating the AWARD characteristics for supporting dynamic reconfigurations.

6.3.1 Scenario 1: Change Task for Modifying the Activity Behavior

This simple scenario illustrates how the AWARD workflow based on basic patterns that was discussed in Section 6.2.3.1 (shown again in Figure 6.18) can be dynamically reconfigured for improving its performance. Even when the Kepler system is used the average of the workflow iteration execution time is always greater than 1 second due to the delay (defined as a *Task* parameter) in the two *Add* activities (*Add1-1sec* and *Add2-1sec*) of the workflow.

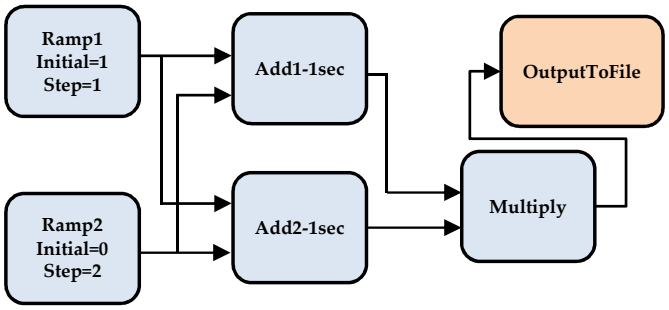


Figure 6.18: Workflow for improving performance by changing *Tasks*

Assuming a long-running workflow with multiple iterations in Kepler if the need arises for improving the workflow execution it is mandatory to develop a new workflow and restart a new execution. However, in AWARD the execution performance can be improved during the workflow execution by submitting a dynamic reconfiguration plan for changing *Parameters* or even the *Task* (algorithm) of any activity. As an example, the average of iteration execution time can be improved by simply changing the *Task Parameters* or even by changing the *Task* algorithms to new algorithms, leading to improved performance of the *Add1-1sec* and *Add2-1sec* activities.

Figure 6.19 illustrates the result of the workflow execution when we dynamically changed the *Task* of the two *Add1-1sec* and *Add2-1sec* activities with a new algorithm that was optimized for a delay of only 0.5 seconds instead of the previous value of 1 second.

This dynamic change was achieved by submitting a reconfiguration plan as presented in Listing 6.7.

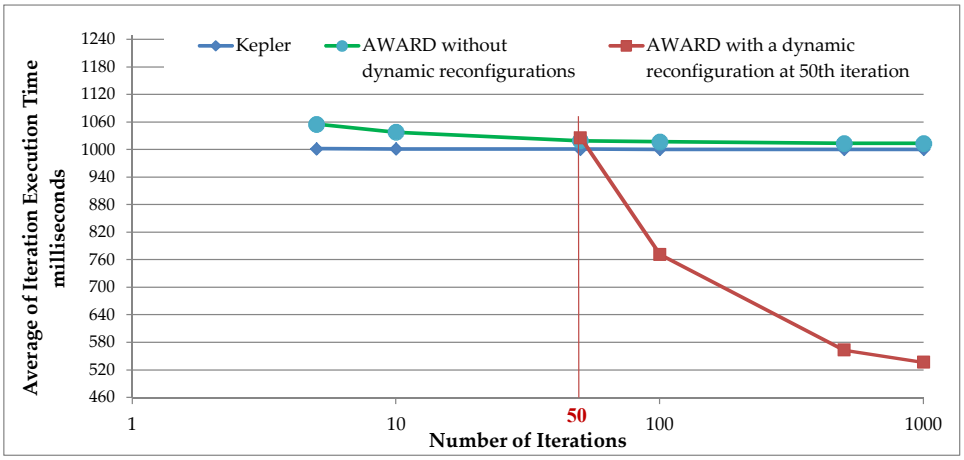


Figure 6.19: Dynamic reconfiguration to improve performance

From Figure 6.19 we can note that in this experiment, the reconfiguration plan was successfully applied at the 50th iteration. As a result, the average iteration execution time decreases after the 50th iteration leading to a reduction of the overall execution time. The reconfiguration plan script uses the *ChangeTask* dynamic operator to change the *Tasks* of the *Add1-1sec* and *Add2-1sec* activities to a new algorithm implemented by the new *Task* named *Add-05sec*.

Listing 6.7: Reconfiguration plan script for changing *Tasks*

```
int RID = BeginReConfiguration(new String[]{"Add1-1sec","Add2-1sec"});
  BeginAwaReConfig(RID, "Add1-1sec");
    ChangeTask(RID, "Add1-1sec", "tasks.Add-05sec");
  int it1 = EndAwaReConfig(RID, "Add1-1sec");
  BeginAwaReConfig(RID, "Add2-1sec");
    ChangeTask(RID, "Add2-1sec", "tasks.Add-05sec");
  int it2 = EndAwaReConfig(RID, "Add2-1sec");
  int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID,new String[]{"Add1-1sec","Add2-1sec"},AgreementSet);
```

Although this example is minimal and very simple it clearly demonstrates the AWARD functionality and feasibility as well as the advantages of performing dynamic workflow reconfigurations.

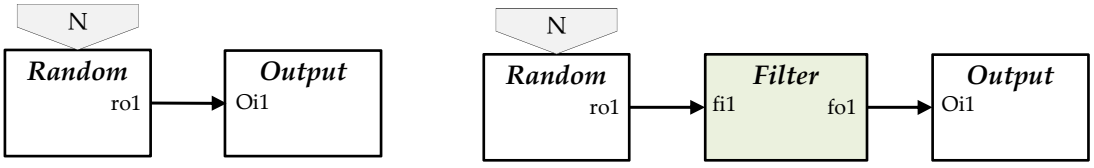
6.3.2 Scenario 2: Change the Structure of a Pipeline Workflow

In scientific data-intensive experiments modeled as long-running workflows, the large size of data involved may require changing or even discarding intermediate data results for avoiding unnecessary computations in the future [Gon+13]. This implies a validation whether data is or not compliant with quality and filtering criteria. However, this can pose a problem due to the difficulties for defining such criteria before the workflow execution. This problem can be overcome by using the AWARD model functionality for monitoring

intermediate data results and the ability for performing structural and/or behavioral workflow reconfigurations.

Therefore this scenario evaluates how dynamic reconfigurations can be used to change the structure of a pipeline workflow by dynamically inserting a new activity for changing the processing of intermediate data sets.

The scenario assumes the AWARD workflow of Figure 6.20(a) with infinite number of iterations. The *Random* activity produces, on its *ro1* output port, an infinite number of tokens as random integer numbers ranging between 0 and a number N, which is specified as an activity parameter. The *Output* activity receives, on its *Oi1* input port, the random numbers and its *Task* displays them in a graphical user interface.



(a) Initial workflow (b) Workflow after applying a dynamic reconfiguration

Figure 6.20: Workflow reconfiguration for data filtering

For illustrating the need to change the workflow behavior consider that the random number 0 is unwanted in the *Output* activity.

The AWARD flexibility to support dynamic reconfigurations allows several approaches for changing the workflow behavior. For instance, the *Task* of the *Random* activity can be changed for not producing the number 0. However, as a more general approach the structure of the workflow can be changed by introducing a new activity named *Filter*, as shown in Figure 6.20(b), which filters the random numbers and replaces the number 0 with a mark, for instance the number -1.

The reconfiguration plan for performing the workflow changes is presented in Listing 6.8, which consists of launching the new *Filter* activity and connecting its *fo1* output port to the *Oi1* input port of the *Output* activity and changing the destination of the *ro1* output port of the *Random* activity for sending tokens to the *fi1* input port of the new *Filter* activity.

For demonstrating the operation of this scenario the *Task* of the *Random* activity generates random numbers between 0 and 9 and the *Task* of the *Filter* activity, named *TaskFilterInteger*, is a Java class with a graphical user interface for showing the random numbers as illustrated in Figure 6.21.

The result of applying the reconfiguration plan that succeeded at the 27^{th} iteration is shown in a dashed box where the random numbers 0 at iterations 33 and 35 are filtered and replaced with the -1 number as received on the *Output* activity.

Listing 6.8: Reconfiguration plan for introducing the new Filter activity

```
int RID = BeginReConfiguration(new String[]{"Random", "Filter"});
    LaunchActivity("FilterConfig.xml", "Filter");
    BeginAwaReConfig(RID, "Filter");
        StartExec(RID, "Filter")
        ChangeOutputLink(RID, "Filter", "fo1", new String[]{"Oi1"});
    int it1 = api.EndAwaReConfig(RID, "Filter");
    BeginAwaReConfig(RID, "Random");
        ChangeOutputLink(RID, "Random", "ro1", new String[]{"fi1"});
    int it2 = api.EndAwaReConfig(RID, "Random");
    int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID, new String[]{"Random", "Filter"}, AgreementSet);
```

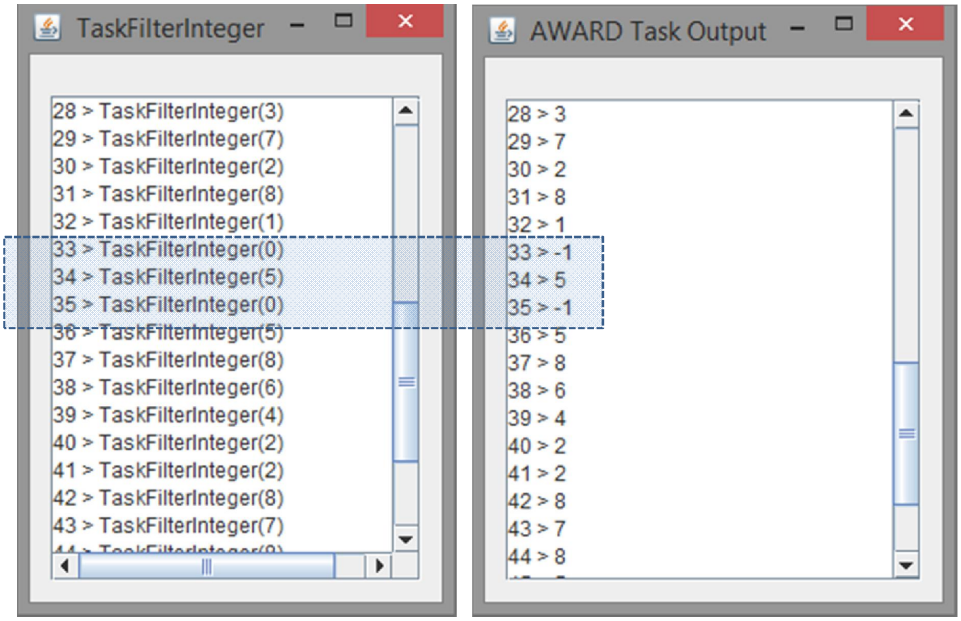


Figure 6.21: Result of introducing the Filter activity

6.3.3 Scenario 3: Change Workflow Structure to Introduce a Feedback Loop

As presented in Section 6.2.4.2 on page 211 the AWARD model expressiveness allows workflow structures with feedback loops to be configured at design time. However, the AWARD support for dynamic reconfigurations allows introducing feedback loops during the workflow execution.

For evaluating this AWARD characteristic we consider a simple pipeline workflow executed during multiple iterations. As presented in Figure 6.22 each activity has a *Task* with a graphical user interface for showing strings with the *Task[@iteration](inputs)* pattern, where *iteration* is the current iteration and *inputs* is a list of tokens received at the activity input ports.

By observing the intermediate results the workflow developer can decide to reconfigure the workflow by introducing a feedback loop as presented in Figure 6.23 where the new *Feedback* activity processes a token received from the *C* activity and sends a feedback

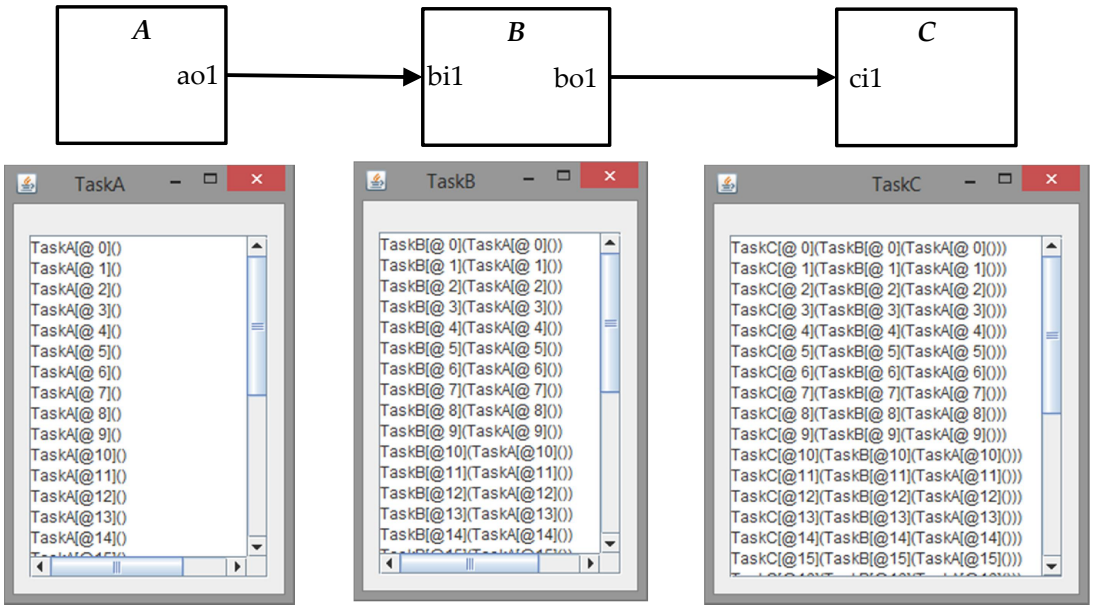


Figure 6.22: Pipeline workflow without feedback loop

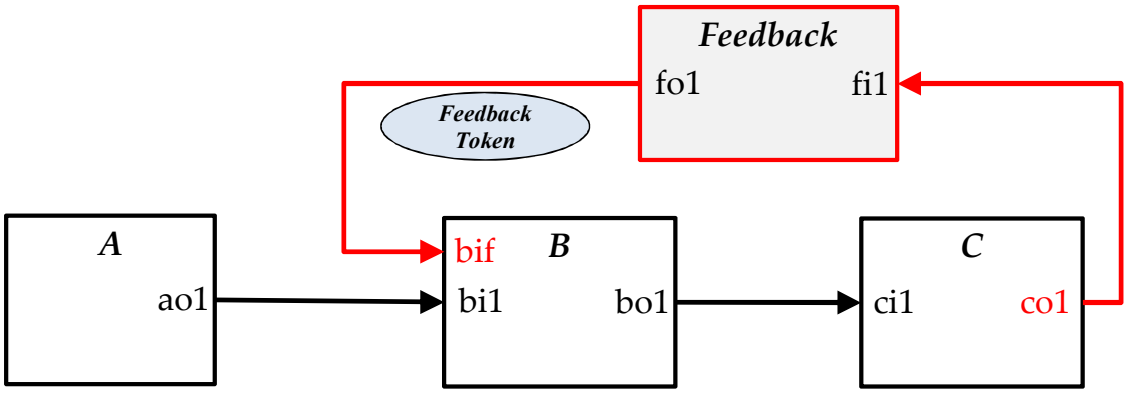


Figure 6.23: Workflow reconfiguration to introduce a feedback loop

token to a new *bif* input port dynamically created on the *B* activity.

The reconfiguration plan for introducing the feedback loop is presented in Listing 6.9, which involves the creation of an input port (named *bif*) and the creation of an output port (named *co1*).

The creation of the new *bif* input port on the *B* activity can require changing the *Task* of the *B* activity as well as the mappings from its input ports to the *Task Arguments*. In the same way the creation of the new *co1* output port on the *C* activity can require changing the *Task* of the *C* activity to produce a new result to be mapped to the new *co1* output port and to be sent as a token to the *fi1* input port of the *Feedback* activity.

In Figure 6.24 the execution result is presented after submitting the reconfiguration plan that succeeded at the 9^{th} iteration where the new *Tasks* of the *B* and *C* activities show the results of introducing the feedback loop.

Listing 6.9: Reconfiguration plan for introducing the feedback loop

```
int RID = BeginReConfiguration(new String[]{"B", "C", "Feedback"});
    BeginAwaReConfig(RID, "B");
       CreateInput(RID,"B", "bif", "EnableFeedback", "java.lang.String", "Iteration");
       ChangeMappingInputs(RID, "B", new String[]{"bi1", "bif"});
       ChangeTask(RID, "B", "tasks.TaskBwith2args");
    int it1 = EndAwaReConfig(RID, "B");
    BeginAwaReConfig(RID, "C");
       CreateOutput(RID, "C", "co1", "Enable", "java.lang.String", "Single");
       AddOutputLink(RID, "C", "co1", new String[]{"fi1"});
       ChangeMappingOutputs(RID, "C", new String[]{"co1"});
       ChangeTask(RID, "C", "tasks.TaskCwithOutput");
    int it2 = EndAwaReConfig(RID, "C");
    LaunchActivity("FeedbackConfig.xml", "Feedback");
    BeginAwaReConfig(RID, "Feedback");
       StartExec(RID, "Feedback")
       ChangeOutputState(RID, "Feedback", "fo1", "EnableFeedback");
       ChangeOutputLink(RID, "Feedback", "fo1", new String[]{"bif"});
    int it3 = EndAwaReConfig(RID, "Feedback");
    int[] AgreementSet=new int[]{it1, it2, it3};
int K=EndReConfiguration(RID, new String[]{"B", "C","Feedback"}, AgreementSet);
```

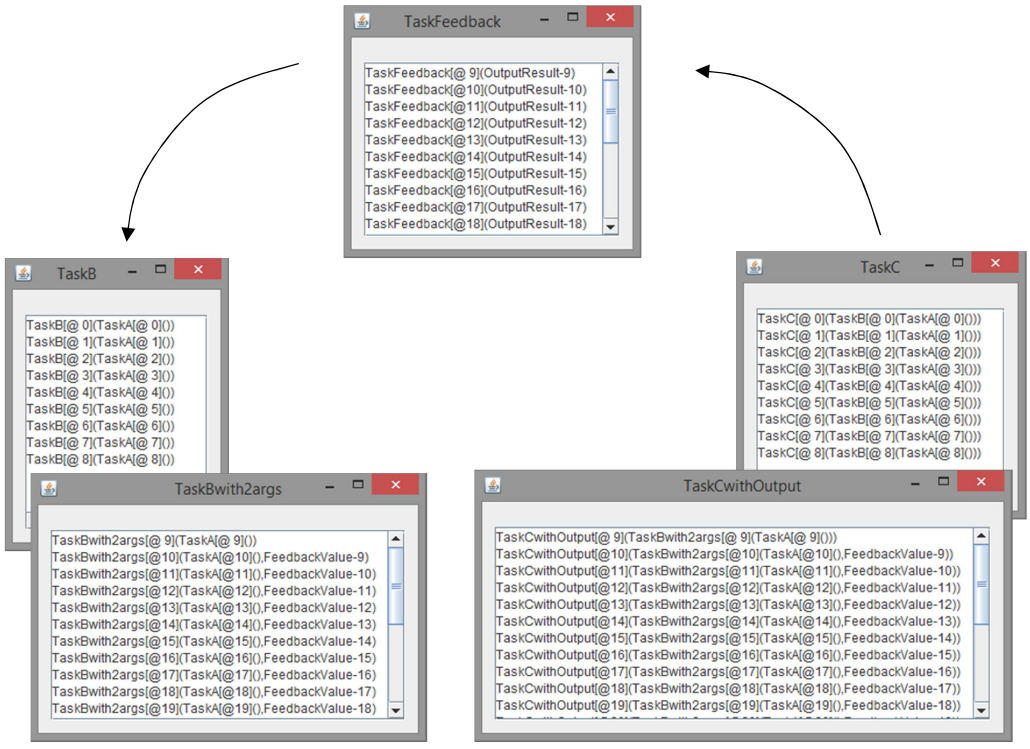


Figure 6.24: Pipeline workflow with a feedback loop after the 9th iteration

6.3.4 Scenario 4: Change Workflow Structure and Behavior for Load Balancing

As presented in Section 6.2.4.3 the AWARD model expressiveness allows workflow structures where an activity can be replicated for load balancing purposes and this can be

configured at design time. However, the AWARD support for dynamic reconfigurations allows introducing the load balancing pattern during the workflow execution.

For evaluating this AWARD characteristic we consider a simple pipeline workflow executed during multiple iterations as presented in Figure 6.25.

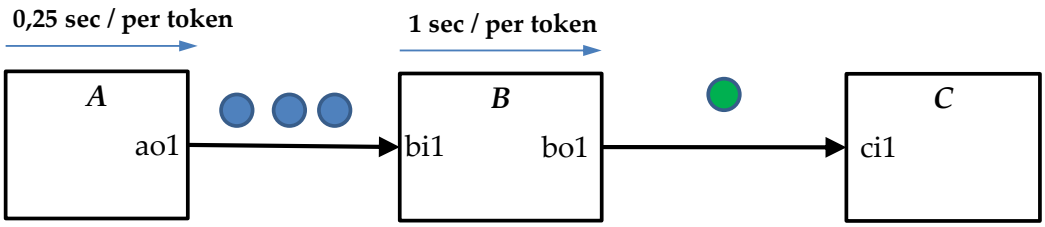


Figure 6.25: Workflow with delayed tokens delivery

The *A* activity produces tokens per each iteration at a pace of 0.25 seconds per token to be processed by the *B* activity and are afterwards delivered to the *C* activity. Given that the *B* activity incurs a delay of 1 second for delivering each token, then after performing multiple iterations the per token delivery time increases for each token. In fact as the producing rate of the *A* activity is four times the pace of the *B* activity the tokens produced by the *A* activity are delayed in the AWARD Space until the *B* activity can consume them.

For evaluating this scenario we developed a token class, depicted in Listing 6.10, for carrying timestamps. Therefore, before issuing tokens the *A* activity marks them with the system time in milliseconds and the *B* activity also marks tokens with the system time before delivering them to the *C* activity.

Listing 6.10: Token class that for carrying timestamps

```
1 import java.io.Serializable;
2
3 public class InterActivityLBToken implements Serializable {
4     public long TaskATime; //timestamp on activity A
5     public String itA; //current iteration at activity A
6     public long TaskBTime; //timestamp on activity B
7     public String balanceName; // Activity B, B1, B2 or B3
8 }
```

As illustrated in Figure 6.26 the per token delivery time (indicated "elapsed" in the Figure 6.26) reach near 1 minute after 75 iterations.

By observation of the intermediate workflow results it is possible to conclude that if we introduce three more replicas of the *B* activity the per token delivery time has a tendency to reach 1 second because for each four tokens produced by the *A* activity there are four instances of the *B* activity for processing them in parallel as illustrated in Figure 6.27.

The AWARD dynamic library operators can be used for preparing a reconfiguration plan to introduce the three replicas of the *B* activity and changing the workflow structure. Additionally the token mode behavior of the *ao1* output port of the *A* activity is

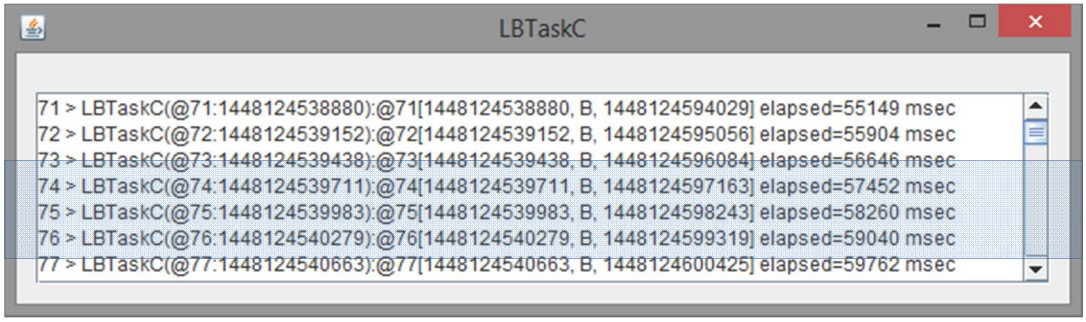


Figure 6.26: Token delivery time reaches near 1 minute after 75 iterations

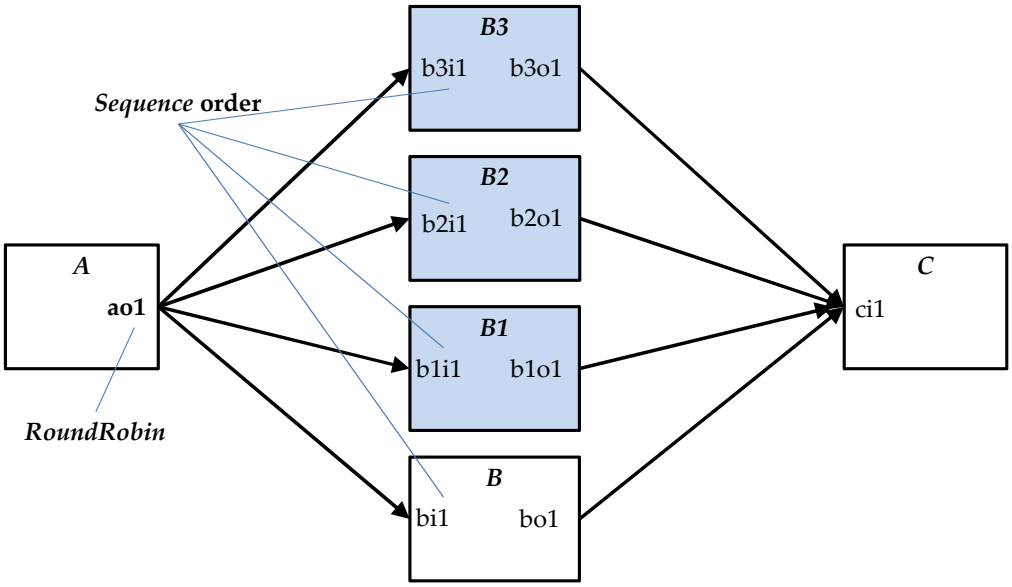


Figure 6.27: The workflow with load balancing on activity *B*

also changed for emitting tokens in round-robin fashion. Also the order mode of the *bi1* input port of the existing *B* activity is changed to receive tokens in *Sequence* mode. Assuming that the *B1*, *B2* and *B3* activities are specified in the *replicasB.xml* file where the *b1i1*, *b2i1* and *b3i3* input ports are already configured with a *Sequence* order mode, the reconfiguration plan for introducing load balancing on the *B* activity is presented in Listing 6.11.

For demonstrating this reconfiguration scenario we used a single standalone computer for launching the initial workflow with the *A*, *B* and *C* activities and for observing the output of the *C* activity and by showing the per token delivery time as presented in Figure 6.28. These per token delivery time are based on the computer current system time in milliseconds.

When per token delivery time exceed 1 minute we submitted the reconfiguration plan of Listing 6.11 that led to the new workflow configuration with an agreement between the involved activities for applying the reconfiguration at the 79^{th} iteration.

Listing 6.11: Reconfiguration plan for introducing load balancing on activity *B*

```
1
2  String[] awaNames=new String[]{"A","B","B1", "B2", "B3"};
3  int[] iters=new int[5];
4  int RID = BeginReConfiguration(awaNames);
5      for (int i=2; i < awaNames.length; i++) {
6        LaunchActivity("replicasB.xml", awaNames[i]);
7        BeginAwaReConfig(RID, awaNames[i]);
8            StartExec(RID, awaNames[i]);
9        iters[i] = EndAwaReConfig(RID, awaNames[i]);
10     }
11     BeginAwaReConfig(RID, "A");
12       ChangeOutputLink(RID,"A","ao1",new String[]{"bi1","b1i1","b2i1","b3i1"});
13       ChangeOutputStrategy(RID, "A", "ao1", "RoundRobin");
14     iters[0] = EndAwaReConfig(RID, "A");
15     BeginAwaReConfig(RID, "B");
16       ChangeInputOrder(RID, "B", "bi1", "Sequence");
17     iters[1]== EndAwaReConfig(RID, "B");
18     int[] AgreementSet=iters;
19 int K=api.EndReConfiguration(RID, awaNames, AgreementSet);
```

Listing 6.12: Reconfiguration plan to launch the *B4* activity

```
1  int RID = BeginReConfiguration(new String[]{"A", "B4"});
2      BeginAwaReConfig(RID, "A");
3          AddOutputLink(RID, "A", "ao1", new String[]{"b4i1"});
4      int it1 = EndAwaReConfig(RID, "A");
5      LaunchActivity("replicasB.xml", "B4");
6      BeginAwaReConfig(RID, "B4");
7          StarExec(RID, "B4");
8      int it2 = EndAwaReConfig(RID, "B4");
9      int[] AgreementSet=new int[]{it1, it2};
10 int K=EndReConfiguration(RID, new String[]{"A", "B4"}, AgreementSet);
```

The results presented in Figure 6.28 [case a)] show that before the 79th iteration the per token delivery time of the *B* activity increases until 65.7 seconds because the processing pace of the *A* activity is four times the pace of the *B* activity. However, after applying the reconfiguration plan at the 79th iteration as shown in Figure 6.28 [case b)] and Figure 6.28 [case c)] the per token delivery time of the *B* activity stabilize close to 63 seconds because it starts receiving tokens near its processing rate and the new *B1*, *B2* and *B3* activities are processing tokens at their pace of 1 second per token.

In order to compensate the remaining delay on the *B* activity a new reconfiguration plan presented in Listing 6.12 is submitted for launching the *B4* activity. This reconfiguration plan, that succeeded at the 492nd iteration, led all five replicas of the *B* activity (*B*, *B1*, *B2*, *B3* and *B4*) to deliver tokens close to its processing rate of 1 second per token as shown in Figure 6.28 [case d)].

Figure 6.29 illustrates the evolution of the per token delivery time on the initial *B* activity showing the points at the 79th iteration and the 492nd iteration where the two reconfiguration plans are applied.

Figure 6.28: Per token delivery time on activity *C*

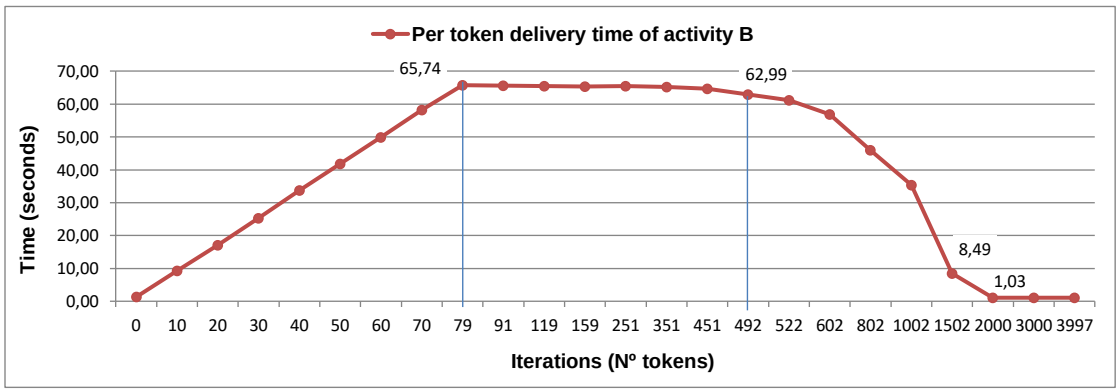


Figure 6.29: Per token delivery time of activity *B*: before and after applying load balancing

This scenario clearly showed the advantage of applying dynamic reconfigurations for optimizing workflow execution times. Additionally it is important to note that after the *B* activity recovery, a third reconfiguration plan can be submitted to remove the *B4* activity. Such reconfiguration plan would simply use the *ChangeOutputLink* operator for changing the *ao1* output port of the *A* activity and a *Terminate* operator to terminate the *B4* activity. Therefore the AWARD characteristics for supporting dynamic reconfigurations can also be used to develop automatic tools for introducing elastic scalability [HKR13] as the capability of dynamically increasing and decreasing the allocation of resources to optimize workload changes during a workflow execution.

6.3.5 Section Conclusions

The reconfiguration scenarios presented demonstrate the functionality and the feasibility of the AWARD model for supporting dynamic workflow reconfigurations using reconfiguration plans implemented with the developed software library (*DynamilcLibaray.jar*), which provides a set of reconfiguration operators. These operators allow structural and behavioral workflow changes that can be used for improving workflow execution times, or for adapting the application functionality to changing requirements.

6.4 Application Cases

The previous sections (6.2 and 6.3) presented various scenarios to demonstrate the flexibility and feasibility of using AWARD for developing scientific workflows. However, the AWARD framework has been effectively used for experimenting further scenarios in the context of the following concrete application cases:

1. An implementation of the *MapReduce* model validated with experiments related to data analytics in the cloud;
2. Workflows for invoking Web Services from external institutions;
3. Recovering from workflow activity faults using dynamic reconfigurations;
4. Workflows with steering by multiple users;
5. The workflow of a text mining application, which is a very significant case because it demonstrates the flexibility and feasibility of the AWARD framework for developing workflows by external users that have no knowledge about the AWARD machine internals.

6.4.1 Case 1: AwardMapReduce Workflow

✧ **Description**

Data analytics applications are currently used for extracting relevant information from large-scale data sets. These applications can be expressed as workflows with multiple activities for data processing, filtering, analysis, combining, and visualization. This requires a flexible composition of the workflow structure and possibly also its configuration depends on the application requirements and the data localization. Therefore to achieve practical results the workflow structure needs sufficient flexibility to use massive parallel processing.

The *MapReduce* programming model has been used for data-parallel processing of large data sets [DG04; Fad+12; Gun+10]. The *MapReduce* developer's view is based on two basic *Map* and *Reduce* functions and the operational view consists of an implicit workflow, which is instantiated by the runtime system with nodes for input data splitting, parallel

evaluation of the *Map* function, partitioning and distributing the generated intermediate data, and parallel evaluation of *Reduce* function.

The *MapReduce* programming model has been applied in data analytic applications to process large data sets in cluster and cloud environments. For developing an application using the *MapReduce* model there is a need to install and configure specific frameworks such as Apache Hadoop [Apa15a] or pay for accessing cloud services for instance the *Elastic MapReduce* in Amazon cloud [Ama12]. However, the original *MapReduce* model and the underlying execution environment lack a flexible support for the configuration and composition of the workflow nodes. Therefore it would be desirable to provide more flexibility in adjusting such configurations according to the multiple phases of data analytic applications.

✧ **Case contribution for the AWARD evaluation**

The motivation for studying this application case was twofold. Firstly to evaluate the functionality and flexibility of AWARD for specifying a specific workflow that implements a well-known programming model. Secondly to evaluate the feasibility of executing such workflow on cloud infrastructures where multiple activities are executed in parallel and can process large data sets.

As a joint work with Carlos Gonçalves and in the experimental context of a text mining application we developed the *AwardMapReduce* workflow for implementing the *MapReduce* model [GAC12]. This work demonstrated how *MapReduce* workflows are supported on top of the AWARD framework, with increased flexibility regarding their configuration and composition within a complex application workflow. It also shows the feasibility for executing the *AwardMapReduce* workflow activities on multiple virtual machines of the Amazon EC2 infrastructure.

The text mining application is expressed as a complex workflow with three phases whose final outcome is to extract the relevant expressions occurring in a natural language *corpus* [GSC15; Sil+99]. The development of this application using AWARD is discussed in Section 6.4.5. Here we discuss an experiment in using AWARD to support the specification and execution of phase 1 of the above application, and explain how this led to the development of the *AwardMapReduce* workflow.

✧ ***AwardMapReduce* workflow specification**

As presented in Figure 6.30, the phase 1 of the application workflow consists of six parallel activities, denoted as *"Phase 1, n-gram"*, with n=1 to n=6. Each activity is responsible for counting all occurrences of n-grams of each given size (n=1, n=6), that is, *"Phase 1, 1-gram"* counts all occurrences of unigrams, *"Phase 1, 2-grams"* counts all occurrences of bigrams, etc.

Counting n-grams in large text files is a well-known problem that has been solved, for example, using the MapReduce model, for each given value of n, in order to explore data parallelism for large data sets.

As illustrated in the expanded workflow node in Figure 6.30, this is achieved by using multiple *Mapper* processes performing the counting of n-gram in separate text partitions,

followed by an aggregation of the partial results, performed by a set of *Reducer* processes.

In this way all nodes in phase 1 can be launched in parallel as separate *MapReduce* workflows, where each is supported by an *AwardMapReduce* workflow in this experiment.

AwardMapReduce has a similar functionality as other *MapReduce* implementations where the functions *Map* and *Reduce* are, respectively, the *Tasks* of *Mapper* and *Reducer* workflow activities of Figure 6.30. However, the fact that *AwardMapReduce* is implemented as an AWARD workflow gives it an increased flexibility in its configuration and *Parameters* definition. When compared to other *MapReduce* platforms, such as Apache Hadoop [Apa15a] the advantage of using *AwardMapReduce* is the AWARD possibility for specifying the same workflow template with different configuration *Parameters* to the *Mapper* activity for counting the n-grams with the different values of n (from 1 to 6) for all nodes of the phase 1.

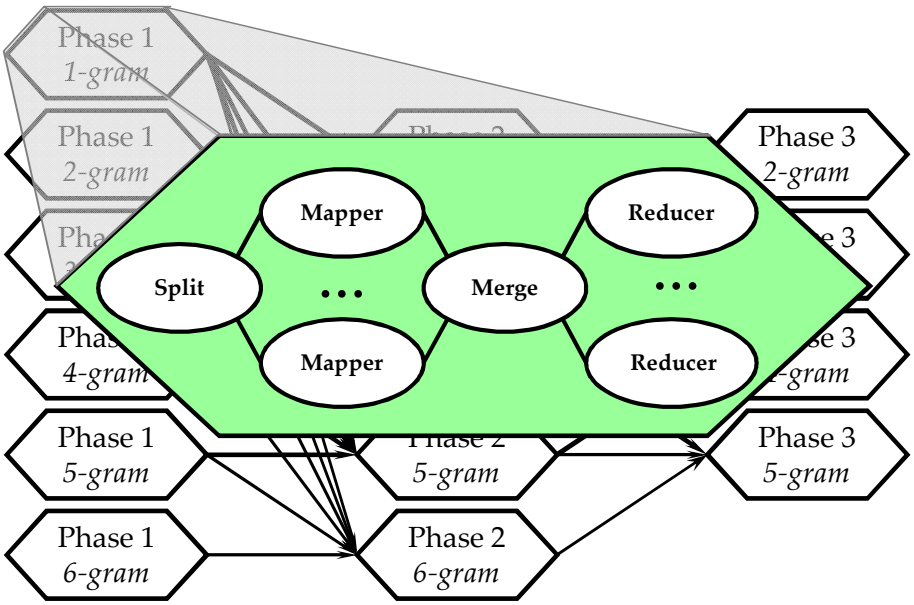


Figure 6.30: Text mining application with a phase 1 modeled as a workflow [GAC12]

✧ ***AwardMapReduce* implementation**

Although the above mentioned experiments were performed in the context of the above mentioned text mining application, we designed the *AwardMapReduce* workflow with the following general requirements:

1. For developing any *MapReduce* application, the programmer only needs to provide the same number of functions as when using other *MapReduce* platforms, typically a function for getting *Records* from the input file, and the *Map* and *Reduce* functions. The *Map* and *Reduce* functions developed in the Java language to run *MapReduce* applications in other platforms, for instance Apache Hadoop, can be reused with minimal modifications;

2. The *AwardMapReduce* workflow is neutral regarding the file systems used. According to the application data and the computational infrastructure the input or output files can be associated with local file systems and distributed file systems;

3. As an AWARD workflow the *AwardMapReduce* workflow can be launched in heterogeneous infrastructures with different operating systems on local networks, clusters or virtual machines in cloud environments.

✧ **The *AwardMapReduce* interface**

The *AwardMapReduce* implementation provides the *AwardMRlib.jar* Java library to facilitate the development of *MapReduce* applications. This software library defines the interfaces and data types presented in Table 6.1 that are similar to other *MapReduce* platforms. Developers only need to use this software library for developing new *MapReduce* applications.

Table 6.1: Overview of *AwardMRlib.jar* library

Base classes: The application dependent Key and Value classes need to derive from the following base classes: *class MRkeyBase implements Serializable, Comparable { }* *class MRvalueBase implements Serializable, Comparable { }*
Class to store Key/Value pairs: The method collect is invoked to store key/value pairs in the shared storage for instance in the *Map* and *Reduce* functions. *class MRcollector { void collect(MRkeyBase k, MRvalueBase v); }*
Interface of the record extractor function: The application dependent class for reading input data must implement this interface where the *getNextRecord* method must return null when no more data records are available. *interface IRecordExtractor { Record getNextRecord(); }* *class Record {MRkeyBase k; MRvalueBase v; }*
Interface of Map function: The application dependent class to process the key/value pairs must implement this interface, and the map method must invoke the method collect of the MRcollector argument. *interface IMap {void map(MRkeyBase k, MRvalueBase v, MRcollector c); }*
Interface of Reduce function: The application dependent class for iterating the values set (v), and applying the reduce algorithm must implement this interface. The final key/value pairs are stored into the shared storage by invoking the *MRcollector* argument. *interface IReduce {void reduce(MRkeyBase k, Iterator<MRvalueBase> v, MRcollector c); }*

✧ **The AwardMapReduce workflow activities**

The *AwardMapReduce* workflow developed in this experiment is depicted in Figure 6.31 with its main activities: *Split*, *Mapper*, *Merger* and *Reducer*. Depending on the size of input data the workflow can be configured to have multiple *Mapper* and *Reducer* activities. Also the workflow can be configured with any additional stages, for example intermediate mergers.

The activities of the *AwardMapReduce* workflow can be spread for distributed execution on multiple computers. Therefore there is a need to share the intermediate data as key/value pairs produced by the *Mapper* activities, merged and sorted by the *Merger* activity and reduced by key at the *Reducer* activities. The *AwardMapReduce* relies on the AWARD Space server for supporting multiple tuple spaces used as a shared storage for storing and retrieving the intermediate data as tuples.

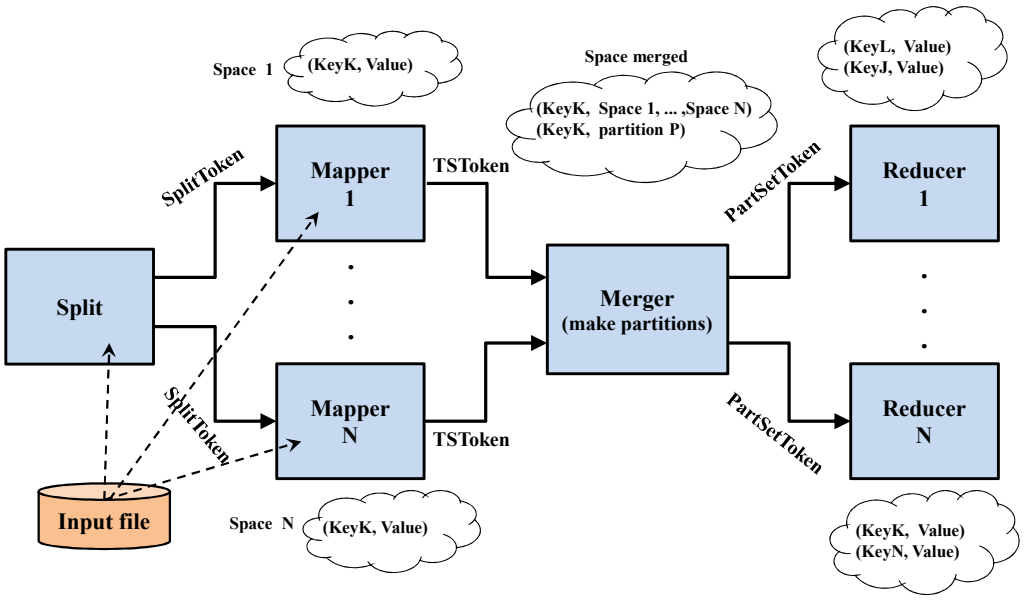


Figure 6.31: The *AwardMapReduce* workflow [GAC12]

Each of the *Mappers*, the *Merger* and each of the *Reducers* has its own tuple space instantiated in the AWARD Space server running on its local computing node. The *Merger* tuple space only contains partitions-related information, to be used by the *Reducers* to locate and read the key/value pairs from the *Mappers* tuple spaces. Therefore there is no copy of key/value pairs from the *Mapper* tuple spaces to the *Merger* tuple space.

The dataflow between the *AwardMapReduce* workflow activities relies on tokens implemented by the Java classes types presented in Listing 6.13. Between the *Split* and the *Mappers*, the *SplitToken* is an object with the file name and the start/end input file offsets for each split. Between the *Mappers* and the *Merger*, the *TSToken* is an object of the *TsServerInfo* type identifying a tuple space (TCP/IP address, TCP/IP port, space name). Between the *Merger* and the *Reducers*, the *PartSetToken* contains information on a *TsServerInfo* object for the tuple space where the information generated by the *Merger* can be found by each *Reducer* and two numbers (*partStart/partEnd*) for defining the sequence of partitions to be processed by each *Reducer*.

In the following we summarize the functionality of the *Split*, *Mapper*, *Merger* and *Reducer* acitivities of the *AwardMapReduce* workflow. More detail of each activity specification can be found in [GAC12].

The *Split* activity calculates the file offset of each file split according to the input file size and a number of splits equal to the number of *Mappers*. The activity sends to each *Mapper* a *SplitToken* (as in Listing 6.13) containing the file name and the split offsets.

The *Mapper* activity invokes the application-dependent *getNextRecord* function according to the *IRecordExtractor* interface (as in Table 6.1) for reading data records from the assigned text file split and invokes the application-dependent *Map* function according to the *IMap* interface (as in Table 6.1) for each key/value pairs (n-gram occurrence) found.

Listing 6.13: Data-flow tokens as Java classes

```
package award.mapreduce;

public class SplitToken implements Serializable { //Split -> Mappers
    public String fileName; // The input file name
    public Long startOffset; // The split start offset
    public Long endOffset; // The split end offset
}

public class TsServerInfo implements Serializable {
    public String hostName; // the IP
    public int tsPort; // the TCP port
    public String tsName; // the name of the tuple space
}

public class TSToken implements Serializable { //Mappers -> Merger
    public TsServerInfo tsmap;
}

public class PartSetToken implements Serializable { //Merger -> Reducers
    public TsServerInfo tsmerg;
    public Integer partStart; // The initial partition number
    public Integer partEnd; // The final partition number
}
```

The pairs are stored into a tuple space identified by the *TSToken* (as in Listing 6.13) to be sent to the *Merger* activity.

The *Merger* activity produces a tuple space with an ordered partition of keys. For each key there is a tuple with a ordered partition number *(key, partitionNumber)* and a corresponding tuple identifying all *Mapper* tuple spaces where the key was processed *(key, Space1,..., SpaceN)*. According to the number of *Reducers* the activity distributes a set of partitions to each *Reducer* by sending a token *PartSetToken* (as in Listing 6.13) with a sequence of key partitions.

The *Reducer* activity produces a tuple space with the final key/value pairs. The activity *Task* uses a thread for each partition number identified in the input token *PartSetToken* (as in Listing 6.13) for performing the following actions: gets a tuple *(keyX, partitionNumber)* from the *Merger* tuple space and using the *keyX* for getting the list of *Mapper* tuple spaces that contain the *keyX* key from the *Merger* tuple space; gets all key/value pairs from the *Mapper* tuple spaces and invokes the application-dependent *Reduce* function according to the *IReduce* interface (as in Table 6.1). In the text mining application the keys are n-grams and the values are the aggregated occurrence numbers of each distinct n-gram.

✧ **Distributed execution of the AwardMapReduce workflow on Amazon cloud infrastructure**

Another goal of these experiments was to evaluate the feasibility of using cloud platforms for executing a data intensive application using the *AwardMapReduce* workflow. In the context of the mentioned text mining application we configured the *AwardMapReduce* workflow to count unigrams in a set of text files of different sizes.

The experiments were performed involving multiple Amazon AWS virtual machines (up to 20 EC2 instances). Given that the *Split*, *Mappers*, *Merger* and *Reducers* activities have different CPU needs during execution we used two distinct EC2 instances types: The Micro type (613 MB memory, 2 EC2 Compute Units, Linux 64-bit, Low I/O performance); and the Large type (7.5 GB memory, 4 EC2 Compute Units, 850 GB instance storage: Linux 64-bit, High I/O performance).

The workflow imposes the following dependencies between activities: The *Split* activity runs first; The *Merger* activity only runs after all the *Mappers* have terminated; and the *Reducers* only run after the completion of the *Merger* activity, therefore after the completion of all *Mappers*. The mapping of workflow activities to EC2 instances was performed as follows:

1. One of the EC2 instances is always dedicated to running the *Split* and *Merger* activities;

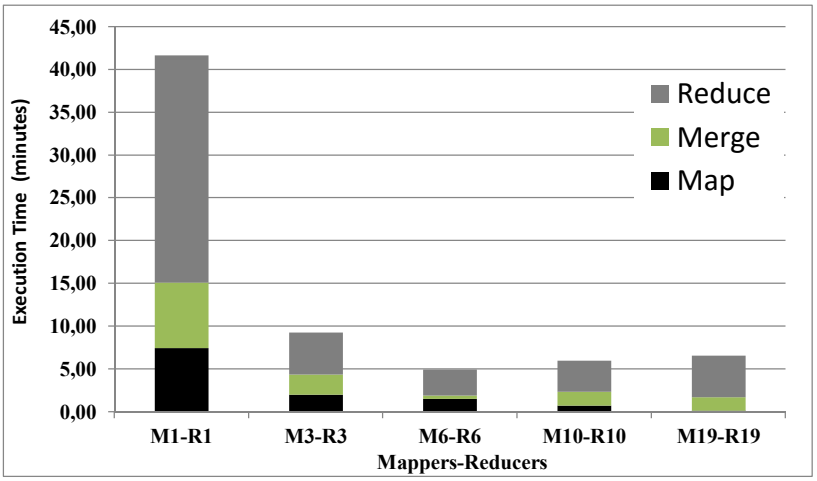
2. Each of the other EC2 instances runs a *Mapper/Reducer* pair;

3. The maximum number of EC2 instances that Amazon allows without special requests is 20. Thus, as an example, the notation M19-R19, used in the charts of Figure 6.32, means that we run 19 *Mappers* and 19 *Reducers* in 19 EC2 instances and in the 20^{th} remaining instance we run the *Split* and *Merger* activities.

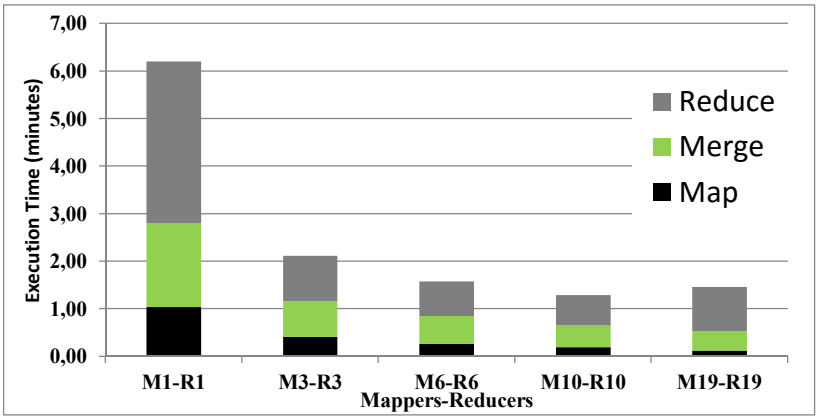
The experiments performed used input files of different sizes (up to 5 Mbyte). Although most of related work usually considers the raw file size only, the number of distinct n-grams (keys) existing in the text input file is particularly relevant as it affects the workload of each workflow activity. In fact distinct n-grams have different number of occurrences. Thus, we present the results of executing an *AwardMapReduce* workflow for a text file with 5709 distinct n-grams when using *Micro* EC2 instances (Figure 6.32(a)) and *Large* EC2 instances (Figure 6.32(b)). The axis "Execution Time (minutes)" is the total execution time for counting the n-grams with n=1, that is unigrams of the text mining application. This time is shown distributed by the parcels of execution time of the *Map*, *Merge* and *Reducer* stages of the *AwardMapReduce* workflow. Both cases in Figure 6.32 show the results of using from 1 to 20 EC2 instances, corresponding to 1 *Mapper* and 1 *Reducer* (M1-R1) up to 19 *Mappers* and 19 *Reducers* (M19-R19).

We consider a speedup definition as $Speedup = \frac{T_{M1-R1}}{T_{Mi-Ri}}$ where T_{M1-R1} is the execution time when using only one *Mapper* and only one *Reducer*, and T_{Mi-Ri} is the execution time when using i *Mappers* and i *Reducers*, with $i = 1, 3, 6, 10, 19$.

In both cases of using EC2 Micro instances (Figure 6.32(a)), and using EC2 Large instances (Figure 6.32(b)) the results allow to conclude that speedup increases with the number of EC2 instances hosting multiple *Mapper* and *Reducer* activities. For instance, when using EC2 Large instances (Figure 6.32(b)) the speedup increases from 3 to 4.8 when comparing the workflow execution with 3 *Mappers* and 3 *Reducers* (M3-R3) to the workflow execution with 19 *Mappers* and 19 *Reducers* (M19-R19).



(a) With Micro EC2 instances



(b) With Large EC2 instances

Figure 6.32: Using EC2 instances for counting unigrams

In terms of the *Map* stage we conclude that the *AwardMapReduce* workflow shows a consistent reduction of the execution time of the *Map* phase when the number of *Mappers* increase.

However, there are different saturation points. For EC2 *Micro* instances, the saturation point is reached close to 6 *Mappers*/6 *Reducers* and for EC2 *Large* instances, the saturation point is only reached close to 10 *Mappers*/10 *Reducers*. These saturation points are related to the following aspects:

1. The bottleneck for accessing multiple tuple space servers. In fact, as the number of *Mappers* and *Reducers* and the associated tuple spaces servers (Figure 6.31) increase, each *Reducer* accesses multiple tuple spaces. Therefore there is room here for improving the mapping of the shared storage as tuple spaces for example by using alternative devices, such as distributed in-memory stores or non-relational data set storage, like Amazon DynamoDB [Ama15a];

2. The *Merge* and *Reducer* activities benefit from using the EC2 Large instances (4 EC2 Compute Units) because their *Tasks* were developed using multithreading. Therefore, the EC2 instance type for *Merger* activity and *Reducer* activities should be chosen with a high number of *Amazon EC2 Compute Units* (ECU).

✧ **Case conclusion**

We have shown that we can execute a particular application phase as an AWARD workflow designed to implement a *MapReduce* model. We also showed that the AWARD framework is flexible to support the implementation and the execution of *MapReduce* workflows using a similar API as used in other *MapReduce* platforms, for instance Apache Hadoop [Apa15a]. Due to the logical abstraction level provided by AWARD model we showed that *AwardMapReduce* workflows are easily mapped onto cloud platforms as Amazon EC2, allowing experiments with multiple computing nodes without dependencies upon specific features, like databases, message queues or file storage.

The experiments also show that the *AwardMapReduce* workflow speedup increases when we use multiple *Mappers* and multiple *Reducers*, thus validating the AWARD flexibility for allowing the development and deployment of alternative *MapReduce* workflow configurations. New workflow activities and stages can be added, for instance to dump the final key/value pairs to output files, or adding new intermediate stages for optimization purposes. As an example, for a *MapReduce* workflow with a higher number of *Mappers* we can easily redesign the workflow with a tree of *Mergers*. For instance, for 16 *Mappers* we can have a first level of 4 *Merger* nodes, a second level with 2 *Mergers* and a final *Merger* to combine and sort the total key/value pairs.

Furthermore, as shown in Section 6.3.4 the AWARD functionality for supporting dynamic reconfigurations can be used to apply load balancing scenarios for dynamically adjusting the numbers of *Mapper*, *Merger*, and *Reducer* activities.

6.4.2 Case 2: Invoking Web Services

✧ **Description**

The success of Service Oriented Architectures (SOA) based on Web Services standards allowed cooperative organizations in multiple science domains to develop Web Services that can be used by anyone anywhere. For example, Taverna [Wol+13] is an open source workflow system for accessing a large number of Web Services in the fields of bioinformatics, astronomy, chemoinformatics, health informatics and others. As an example of a Web Service, the *Protein Identifier Cross-Reference Service* (PICR), described in [Cot+07] and available in [EBI12] offers the possibility of obtaining a protein sequence given an protein identifier by allowing access to a mapping algorithm that uses over 70 distinct databases organized as UniProt Archive [Uni12] as data source. Thus, due to the importance that any workflow system supports the access to the third party Web Services we developed an application case for accessing the above mentioned PICR Web Service for getting protein sequences.

✧ **Case contribution for the AWARD evaluation**

This application case, already presented in [AGC14], demonstrates the functionality and feasibility of the AWARD model to invoke third party Web Services. The AWARD model is neutral concerning the internals of the activity *Tasks*. Thus any activity *Task* can transparently invoke one or more Web Services.

✧ **The workflow specification**

The implemented workflow is depicted in Figure 6.33. For each iteration, the *AwaPICR* activity invokes the Web Service to map a protein identifier received from its *Pws-IN* input port for producing two results: a *UniParc Protein Identifier* (UPI) at the *Pws-O1* output port and the protein sequence at the *Pws-O2* output port. The *AwaPICR* activity specification includes a list of initial *Parameters* including the URL of the PICR Web Service, a list of data base names and a taxonomy identifier (ID) to limit the mappings to the *Homo sapiens* species.

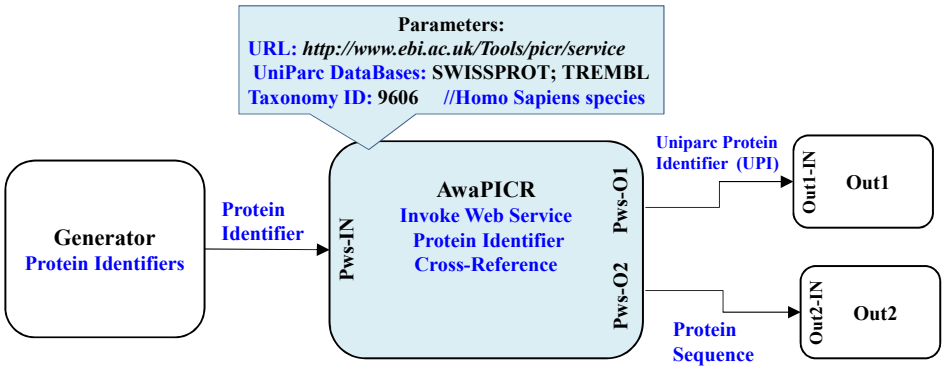


Figure 6.33: Workflow that invokes PICR Web Service to map protein sequences

Listing 6.14 illustrates a view of the workflow specification XML file with details of the AWA activity that invokes the PICR Web Service and the corresponding definition of the input and output ports.

✧ **Implementation**

Listing 6.15 illustrates the relation between the specification of AWARD activities and the implementation of an activity *Task* that invokes the *Protein Identifier Cross-Reference Service* (PICR) Web Service. In the workflow specification details related to the PICR are specified as *Parameters*, for instance the location (URL) of the Web Service. Note the programming simplicity to get the *Arguments* and *Parameters*, and the possibility to return many objects that will be mapped to the activity output ports. The Java package *uk.ac.ebi.picr.model* containing the method *getUPEntry* and the *UPEntry* type was automatically generated by parsing the Web Service WSDL (Web Service Definition Language). For instance, in Netbeans (the Java integrated development environment used in these experiments) the programmer only needs to supply the Web Service URL to generate a package which contains the wrapper classes and the data types for invoking the Web Service.

Listing 6.14: The AWA activity specification that invokes the PICR web Service

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AwardWorkflow>
   <workflowName>workflow Invoke Protein PICR Web Service</workflowName>
   <MaxIterations>9</MaxIterations> <!-- Number of Iterations -->
   <AdditionalLibs>D:\AWARD\Libs\TasksInvokeProteinWS.jar</AdditionalLibs>
   <Awa> <!-- Generate from a file a sequence of nine Proteins IDs --> </Awa>
   <Awa> <!-- Invoke the PICR Web Service -->
       <ControlUnit>
          <InitialState>Idle</InitialState>
          <RulesFileName>TaskBasicRules.clp</RulesFileName>
       </ControlUnit>
       <name>PICR</name>
       <input>
          <name>Pws-IN</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <orderMode>Iteration</orderMode>
       </input>
       <output>
          <name>Pws-O1</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>Single</modeToken>
          <sendTo>Out1-IN</sendTo>
       </output>
       <output>
          <name>Pws-O2</name>
          <state>Enable</state>
          <tokenType>java.lang.String</tokenType>
          <modeToken>Single</modeToken>
          <sendTo>Out2-IN</sendTo>
       </output>
       <Task>
          <parameters>
             <!-- Url of the Web Service -->
             <par>http://www.ebi.ac.uk/Tools/picr/service</par>
             <!-- list of databases to map proteins -->
             <par>SWISSPROT;TREMBL</par>
             <!the taxonomy ID to limit the mappings to Homo Sapiens species -->
             <par>9606</par>
          </parameters>
          <SoftwareComponent>
             <mappingArgs>
                <arg> <idxArg>0</idxArg> <inName>Pws-IN</inName> </arg>
             </mappingArgs>
             <taskImplementationType>
                 tasksinvokeproteinws.TaskInvokeProteinsWS
             </taskImplementationType>
             <mappingResults>
                <result> <idxRes>0</idxRes> <outName>Pws-O1</outName> </result>
                <result> <idxRes>1</idxRes> <outName>Pws-O2</outName> </result>
             </mappingResults>
          </SoftwareComponent>
       </Task>
    </Awa>
    <Awa> <!-- Activity Out1 with GUI interface to display UPI results --> </Awa>
    <Awa> <!-- Activity Out2 with GUI interface to display Protein Sequence --> </Awa>
</AwardWorkflow>
```

Listing 6.15: Pseudo-code of the AWA *Task* that invokes the PICR Web Service

```
package tasksinvokeproteinws;

import awardtaskinterface.AwaContext;
import awardtaskinterface.IGenericTask;
import java.util.ArrayList;
import java.util.List;
import uk.ac.ebi.picr.model; // generated by adding the web Service reference

public class TaskInvokeProteinsWS implements IGenericTask {

   public Object[] EntryPoint(Object[] args, Object[] params) {
      String proteinID=(String)args[0];
      String queryIdVersion = null;
      //a list of all databases to map to
      List<String> searchDB = new ArrayList<String>();
      searchDB.add(params[1]);
      searchDB.add(params[2]);
      //the taxonomy ID to limit the mappings to H. Sapiens
      String taxonomyID = params[3];

      //get the UPEntry from method getUPIForAccession of the PICR web service proxy
      UPEntry entry = getUPEntry(proteinID, null, searchDB, taxonomyID, true);

      Object[] Results=new Object[2]; Task return two results
      If (entry == null) {
        Results[0]="UPI: No mappings for the protein";
        Results[1]="Sequence: No mappings for the protein";
      } else {
        Results[0]=entry.getUPI();
        Results[1]=entry.getSequence();
      }
      return Results;
   }
}
```

✧ **Operation of the workflow**

The workflow execution result is visualized in the *Out1* and *Out2* activities as illustrated in Figure 6.34. For each of the nine workflow iterations the Out1 activity shows the results of the *UniParc Protein Identifier* (UPI) and the *Out2* activity shows the protein sequence for a distinct protein identifier generated by the *Generator* activity.

✧ **Case conclusion**

This application case demonstrates the AWARD functionality and feasibility for allowing workflows whose activities can access third party Web Services. In a transparent way the invocation of Web Services is encapsulated into the activity *Task*. This allows access to Web Services to be performed in the same way as is common in Java programming techniques and Java development environments.

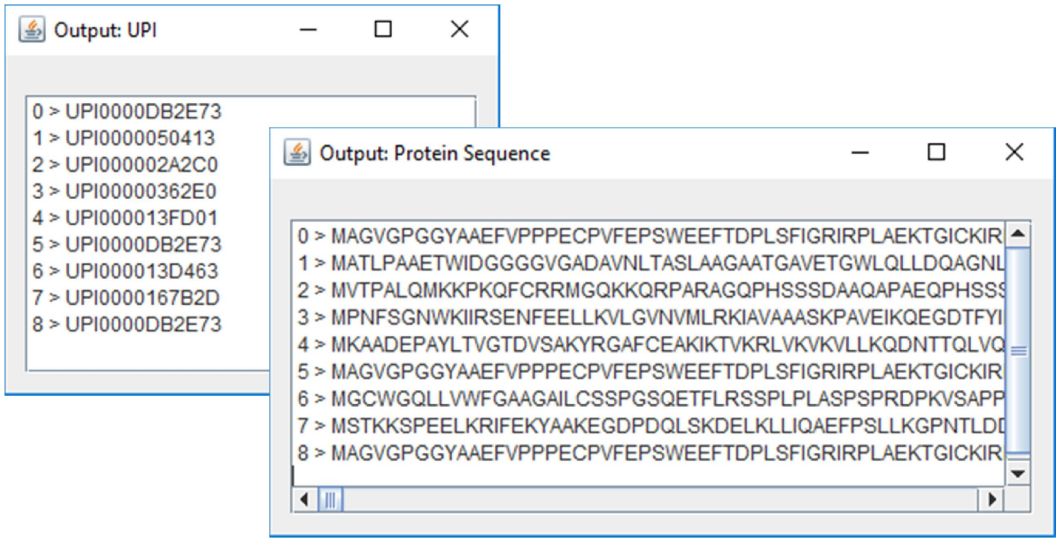


Figure 6.34: Workflow result for nine protein IDs

6.4.3 Case 3: Dynamic Reconfiguration Towards Fault Recovery

✧ **Description**

Workflows are characterized by multiple, eventually infinite number of iterations for processing data sets in multiple activities according to the workflow graph. Some of these activities can invoke cloud services often unreliably or with limitations on their quality of service provoking faults.

After a fault has occurred the most common approach requires restarting of the entire workflow which can lead to a waste of execution time due to unnecessarily repetition of computations.

This application case, already presented in [AC13], discusses how the AWARD framework supports recovery from activity faults using dynamic reconfigurations. This is illustrated through an experimental scenario based on a long-running workflow where an activity fails when invoking a cloud-hosted Web Service with a variable level of availability. On detecting this, the AWARD framework allows the dynamic reconfiguration of the corresponding activity to access a new alternative Web Service, and avoiding restarting the complete workflow anew.

As shown in Figure 6.35, when a fault is detected in some workflow systems the only possibility is to restart the workflow leading to waste of time *(Tb+Tf)*. However, mainly when the *Tb* elapsed time before the fault is significant, for instance hours, it should be possible to decide to reconfigure the workflow in order to recover from the fault.

The time intervals for executing a workflow considering the occurrence of a fault and a reconfiguration towards the fault recovery and its relations are illustrated in Figure 6.35, and described in the following:

- *T* is the foreseen (estimated) execution time;

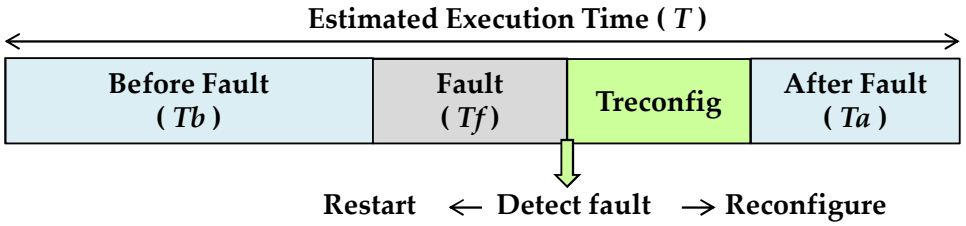


Figure 6.35: Execution time with reconfiguration after a fault

- Tf is the elapsed time in fault;
- *Treconfig* is the time to reconfigure a workflow;
- *Trestart* is the time to completly restart a workflow;
- The elapsed time before fault (Tb) is less than T;
- The foreseen time after fault (Ta) is less than T.

Thus, if no faults occurred (Tf is equal to 0) the workflow execution time (*Texec*) is equal to T, that is, $Texec = T = Tb + Ta$. If after a fault the workflow needs to be restarted without reconfiguration $Texec = Tb + Tf + Trestart + T$; Otherwise the workflow execution time with dynamic reconfiguration after one fault (*TexecR*) is $TexecR = Tb + Tf + Treconfig + Ta$. Both *Trestart* and *Treconfig* times are dependent on the structure of the workflow, for example, the number of activities involved. However, *Trestart* is deterministic and composed of two components: i) *Trestart1*: The time to map activities to the computing nodes; and ii) *Trestart2*: The time of launching the activities on the computing nodes.

The *Trestart* time can be significant in the case of remote executions for instance on a cloud infrastructure.

Assuming scenarios where a single activity is faulty, *Treconfig* time can be composed of the following components: i) *Treconfig1*: Time to observe the fault details and to select the reconfiguration alternatives for a recovery strategy, such as change the $Task$ associated to the activity or change some $Parameters$, for instance a Web Service location. This time is not deterministic but it can be small if the actions to reconfigure the activity are previously known; ii) *Treconfig2*: Time to apply the recovery actions by using a tool to invoke a reconfiguration plan to the faulty activity; and iii) *Treconfig3*: Time to complete the agreement synchronization plus the processing time of the reconfiguration operators.

Comparing *Trestart* and *Treconfig* is not easy because these times are determined by each specific workflow scenario. However, considering that $Ta < T$ and assuming scenarios where the *Treconfig* has the same order of magnitude as *Trestart*, we conclude that the execution time with reconfiguration (*TexecR*) is always less than the execution time without support for reconfigurations (*Texec*).

Furthermore, as long as *Tb* tends to *T* then *Ta* tends to zero and excluding *Trestart*, *Treconfig* and *Tf*, we can then conclude that $Texec \approx 2T$ and $TexecR \approx T$.

This allows us to conclude that an effective recovery strategy based on dynamic reconfiguration plays a critical role to enable the efficient execution of long-running workflows allowing to save substantial elapsed execution time.

✧ Case contribution for the AWARD evaluation

This application case demonstrates how AWARD can be used for modeling and experimenting with real workflow scenarios involving the access to cloud services. The AWARD flexibility is exploited to manage faults by enabling fault detection using the logging information produced during the workflow execution and performing the recovery by applying dynamic reconfigurations without changing the workflow design. This application case also demonstrates that fault recovery, by avoiding the need of restarting the execution of long-running workflows, allows saving significant elapsed execution time.

In AWARD long-running workflows with large number of iterations the activities are enabled to invoke their *Tasks*. For each iteration, *Tasks* are invoked with data *Arguments* from the inputs and a list of initial *Parameters*. Then in some situations unexpected faults can be raised, for example the *Task* algorithm does not expect some combinations of *Arguments* and *Parameters*, or the *Task* requires external resources which may have been unavailable.

In the presence of these scenarios AWARD catches the fault and produces log information reporting the fault occurrence (Section 5.9.1 on page 185). Logs can be analyzed by the user for establishing a strategy to solve the problem and recover the activity from the Fault state and later resuming the execution. The strategy can be more or less complex. The user can modify the workflow structure by replacing the faulty activity with one or more activities, or can change the complete *Task* (algorithm), or can simply modify some *Parameters*, for instance, by redirecting the *Task* to new resources. Once the strategy is defined the user can specify an automatic procedure to handle similar occurrences in the future.

✧ The workflow specification

Nowadays in the context of social networks, blogs and discussion forums, millions of small set of sentences of text, called posts, are produced using multi-cultural languages. These posts can be submitted to text analytics processing in order to extract relevant information such as statistics about occurrences of specific words, for instance personal names, entity recognition, brands references, etc. Workflows are an adequate way to develop such applications allowing the composition of different steps involved on text analytics processing.

We developed an AWARD workflow, presented in Figure 6.36, that supports the continuous (infinite number of iterations) processing of millions of posts in multi-cultural languages and produces a continuous ranking of relevant words that appear in posts.

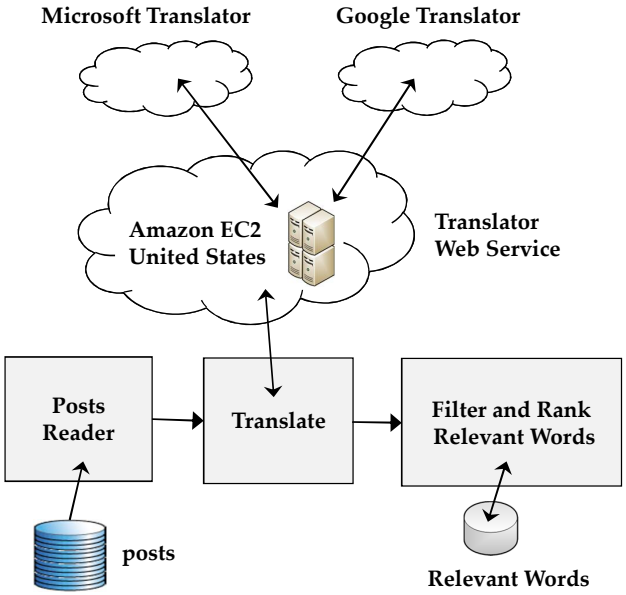


Figure 6.36: AWARD workflow to process posts and rank relevant words

✧ **Implementation**

The *Posts Reader* activity scans a file system directory for searching posts stored in text files with arbitrary filenames and an extension indicating the cultural language, for example *facebook535.eng*, stores an English post originated in Facebook.

For each iteration the *Translate* activity receives a post and the cultural language and if it is not Portuguese, then it invokes a *Translator Web Service* hosted in Amazon EC2. This Web Service is a wrapper that we developed in order to allow: i) Using multiple applications with an independent interface from the existing and widely used translation services; ii) Using translation service providers that offer better prices and good reliability; iii) Using alternatives to ensure better availability. Currently we can use the *Bing Microsoft Translator service* [Mic15] and *Google translator service* [Goo15].

Because the Google service is only available as a paid service we use mainly the *Bing Microsoft Service* as a free of charge service. However, the *Bing Microsoft service* provokes SOAP fault messages when throttling policies with quota limits are reached. This introduces difficulties to have reliable long-running workflows because the *Translator Web Service* hosted in Amazon and consequently the *Translate* activity depends on *Bing Translator* reliability and quality of service. These unreliability characteristics are transferred to the *Translator Web Service* hosted in Amazon and consequently the *Translate* activity is critical due to the possibility of failures caused by multiple situations: i) Quality of service (quotas, throttling, enforced throughput limits); ii) Service unavailability provoked by loss of connectivity or timeouts to the Amazon EC2 infrastructure; crashes in virtual machines (EC2 instances); and iii) possible crash of the HTTP server (Apache Tomcat) that hosts the *Translator Web Service*.

The *Filter* and *Ranking Relevant Words* activity tokenizes the sentences in Portuguese

words and discards the Portuguese words not registered in the relevant words repository. If the Portuguese word is registered, the rank of this word is updated according to the rules defined in the repository. For example for each *N* word occurrences the rank increases by one point.

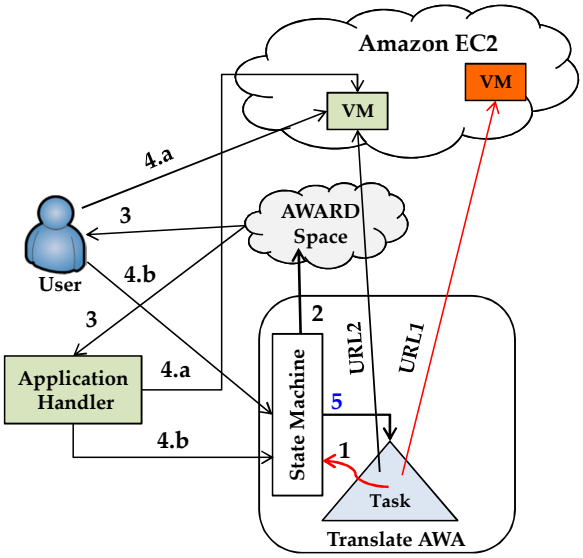


Figure 6.37: Sequence of actions to detect and to recover from failures

The sequence of actions that illustrate how AWARD supports fault detection and recovery using dynamic reconfigurations is depicted in Figure 6.37 and is described as follows:

1. The *Translator Web Service*, using the *URL1* throws a fault inside the *Task* of the *Translate* activity;
2. The *State Machine* of the faulty AWA activity catches the fault and logs the fault context as tuples into the AWARD Space;
3. A user or an application tool inspects the log tuples and analyzes the type of fault;
4. Depending on the fault type an appropriate strategy is performed. In this case we can have manual operation involving user intervention, or an automatic procedure, both consisting of the following steps:
   a) Create a new virtual machine instance, hosting the *Translator Web Service* and obtaining a new *URL2*;
   b) Submit a dynamic reconfiguration plan with the *ChangeParameters* operator to change the parameter to the new *URL2*.
5. The *State Machine* of the *Translate* activity processes the dynamic reconfiguration plan and resumes the execution using the new Web Service located through the new *URL2*.

✧ **Operation of the workflow**

The Virtual Machines used to run the *Translator Web Service* can be of any EC2 instance type, launched from the *Amazon Machine Image* (AMI) [Ama13] (ID=ami-3584f75c), configured with Linux and the Apache Tomcat as the HTTP server to host the *Translator Web Service* at 40001 port.

We made several experiments with the *Translator Web Service* hosted in Amazon US East (N. Virginia) and EU (Ireland) datacenters. To avoid high costs we always used a less expensive EC2 instance type (t1.micro; 64 bit; 1 ECU; Memory 615 MByte; Very Low Network performance). This type of EC2 instance proved to be good for our experimentation, because the low network performance combined with the quota limitations on *Bing Microsoft Translator* provoked frequent failure situations on *Translate* activity, allowing testing several dynamic reconfigurations to recover from these faults.

To run the workflow we used a local cluster with a disk storage (1 Tbyte) shared by all computer nodes all connected by one Gbit Ethernet switch. Each computer node is a dual core Pentium with 4GByte RAM. Post files and the database with relevant words were located in disk. Each workflow activity (*Posts Reader*; *Translate* and *Filter and Ranking*) was mapped to one computer node. The AWARD Space server was run on another computer node.

The experiments conducted were based on a set of posts with sizes presented in Table 6.2. The workflow processed 34762 posts in Portuguese, English and Spanish languages where roughly 8% are posts in English and Spanish languages leading to 50893 translated words.

Table 6.2: Sizes of posts processed

Posts	Text lines	Total Words	Translated Words
34762	83428	710021	50893

Although we have not conducted extensive scalability and throughput experiments, Figure 6.38 presents the elapsed execution time *versus* the number of translated words for processing the posts presented in Table 6.2. The results of two experiments are depicted: i) Manual Reconfiguration; ii) Automatic Reconfiguration.

Both show the elapsed execution time for successive faults and corresponding recovery actions by the user and by an automatic procedure. After 9594 words translated (see Fig. 6.39) the first fault happened with the message *"AppId is over the quota!"*.

On manual reconfiguration, the application user detects this fault by inspecting the log information, as partially presented in Figure 6.39. Then the user launches manually a new EC2 instance with a new *Translator Web Service* hosted in the new *URL2* location and reconfigures the faulty *Translate* activity by using a reconfiguration plan with the *ChangeParameters* operator for changing the *URL1* to the new *URL2*.

As a user, we spent 8 minutes to do this reconfiguration (see Figure 6.38). This time includes the time spent by the user analyzing the log information, plus 2 minutes and 20

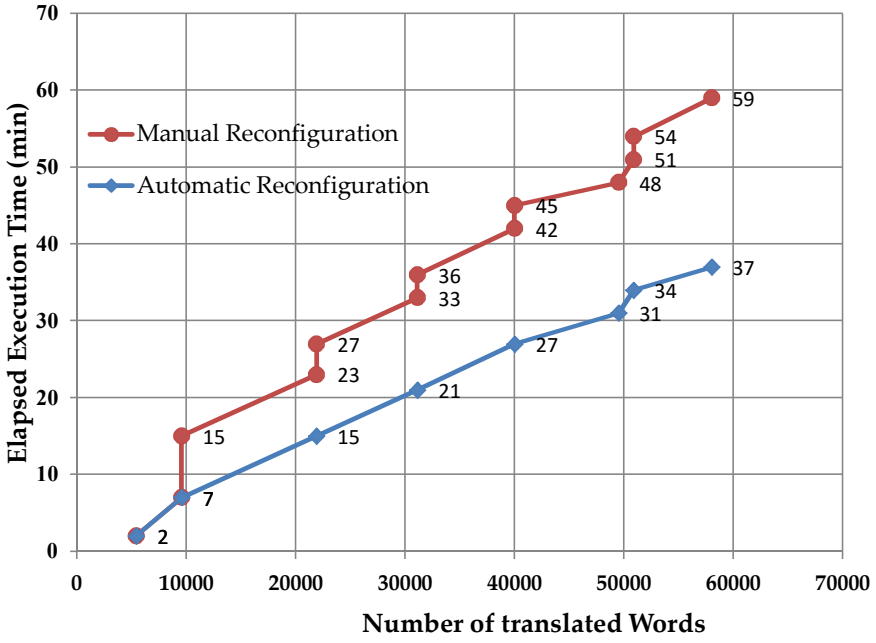


Figure 6.38: Manual *versus* automatic reconfiguration, with faults marked

seconds to create a new virtual machine from a pre-configured *Amazon Machine Image* [Ama13]. The above also includes a negligible time (average of 13 milliseconds) for the *Application Handler* to submit the reconfiguration plan by injecting tuples into the AWARD Space through the dynamic library (*DynamicLibrary.jar*) and for its processing by the *Dynamic Reconfiguration Handler* of the *Translate* activity.

In the automatic reconfiguration procedure, the fault scenario is already known and the user has already created the new EC2 instances. As a result, less time is spent by the user to submit dynamic reconfigurations plans (3 minutes).

Furthermore once this fault pattern was properly identified, the user prepares an external *Application Handler* (as indicated in Figure 6.37 on page 245) to perform automatically the fault handling and the recovery by using the dynamic reconfigurations API.

This corresponds to the following actions:

1. Manage a pool of EC2 instances with the *Translator Web Service*;

2. Wait for log tuples in the AWARD Space indicating that the activity was in fault state;

3. Invoke a dynamic reconfiguration sequence with the *ChangeParameters* operator to change the *Translate* activity parameter, so that the activity *Task* uses a different location (*URL2*).

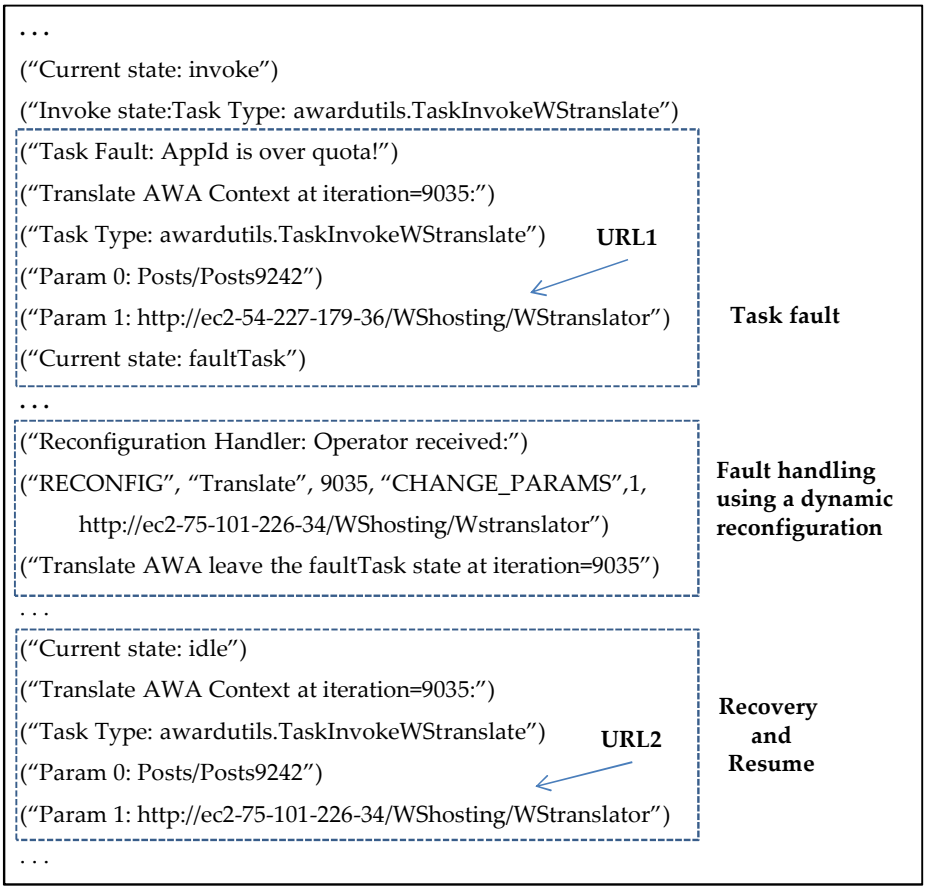


Figure 6.39: Partial snapshot of the log information in the AWARD Space

This automatic procedure allows a substantial improvement in the elapsed execution time compared to the manual user intervention (22 minutes to translate near 60000 words) (see Figure 6.38).

✧ **Case conclusion**

The results of the experiments clearly demonstrate that for long-running workflows, when the elapsed execution time before a fault (*Tb*) is significantly greater than the reconfiguration time (*Treconfig*), there is a clear advantage of recovering by using dynamic reconfigurations.

This application case demonstrates the applicability of AWARD characteristics for supporting dynamic workflow reconfigurations to recover from faults in long-running workflows. We consider this is a distinctive contribution comparing to the existing approaches for failure handling in workflow systems.

We also demonstrated through a real scenario how a substantial elapsed execution time (hours or days) can be saved when a long-running workflow is accessing faulty cloud services.

In the presented scenario we need third party services so it is not easy to have multiple alternative solutions. However, in order to improve reliability or quality of service in other

scenarios we can use the AWARD support for other kind of dynamic reconfigurations. For example: i) Change completely the *Task* type (new Algorithm), for instance to invoke services hosted in private clouds; ii) Replace the *Task* with a new *Task* dependent directly on the Microsoft/Google translator services; iii) Introduce new activities for instance to filter some data sets; and iv) Introduce new activities for load balancing purposes.

During our experiments we observed high performance variations when using virtual machines hosted in Amazon USA or Ireland datacenters depending on the time of day. So even without faults it would be easy to develop an application that depending on the time of day could automatically and dynamically reconfigure workflows in order to use resources with improved performance.

This distinctive characteristic of the AWARD approach unifies dynamic reconfigurations to support structural and behavioral workflow changes on multiple scenarios as well as to support fault recovery.

Despite the scenario presented in this paper is focused on a single activity we would like to note that multiple activities can be independently monitored and reconfigured by distinct users or different application handlers.

6.4.4 Case 4: Reconfiguration and Steering by Multiple Users

✧ Case description

Scientific cooperation is usually exploratory and unpredictable requiring a continuous interaction and intervention by multiple users. For instance, the requirements of a scientific application may not be completely defined at the beginning of an experiment thus forcing scientists to discuss and decide on subsequent actions, which can be based on intermediate experimental results, or/and based on newly gathered advice by other expert scientists. Furthermore typically scientific collaborations in Big Data applications exhibit a data-centric pattern where intermediate data with small sizes are extracted from large data sets and then used by other applications along the collaboration process. As such scientific scenarios rely on the storage of Big Data in distributed repositories spread in multiple data centers, there are several currently open issues. On the one hand the size and privacy of data preclude their movement between data centers. On the other hand some scientific experiments require the processing of distributed data that are stored on multiple repositories [Ben+12]. For instance despite the increasing resource capacity made available by many cloud providers in terms of data storage and processing power, in many application scenarios it may not be adequate to develop a centralized approach for data storage and processing on a cloud provider [Arm+09], [MTB13], [Vah+13]. Another issue is related to the current lack of decentralized execution models allowing the composition of distributed and interactive tasks contributing to a common problem-solving goal and also supporting user steering where each user is responsible for executing, monitoring and dynamically reconfiguring specific tasks without the need to restart or change the activities by other users. Most of the existing approaches are based on a centralized

execution engine, although they may still allow the composition of distributed services [Plo+13], [Wol+13] or the support for allocating tasks into the appropriate distributed resources according to the application requirements [Xia+12]. Other approaches are based on forms of decentralized control for example using peer-to-peer networks [BB11]. However, typically such approaches do not support the dynamic reconfigurations of the scientific experiments with steering involving multiple users [Mat+15]. Many scientific experiments require international cooperation with multiple users at a global scale. For instance, the weather forecasting and the evaluation of the impacts of climate global changes spread beyond national boundaries, necessarily forcing a worldwide collaboration between scientists mainly when extreme climate events affect the world society in terms of agriculture, aviation, shipping, water issues, etc. For instance, when extreme climatic conditions are occurring in the North of America, also having influence upon Europe storms that cause enormous damage in coastal regions of the Atlantic countries such as Portugal, Spain, France, Ireland and Great Britain. Due to this, severe storm warnings and climate monitoring and other geographically distributed phenomena are highly dependent upon international information exchange and collaborations. To foster this international cooperation, organizations such as the *World Meteorological Organization* (WMO), the *Network of European Meteorological Services* (EUMETNET) and, among others, the *Bureau of Meteorology of Australian Government* are continuously observing, collecting and analyzing Big Data sets related to land temperatures, rainfall, atmosphere winds and ocean tides in order to predict the weather and broadcast alerts as soon as possible. Forecasters and scientists from universities and international institutions are constantly processing these data sets gathered from their local sources, such as weather balloons, satellites, rain gauges, barometers, thermometers, radars, and from the measurements taken by ships and aircrafts, aiming at predicting the weather and ocean conditions (such as waves, swells, currents, salinity levels and temperatures) for up to seven days. However, to enhance such predictions it is important to continuously receive inputs from other regions of the globe, mainly exceptional phenomena that could drastically affect the weather in the next days.

Figure 6.40 depicts a scenario of a multinational scientific cooperation that among others can be applied to meteorological sciences. Such scenario was also presented in our publication [AC14] and the subjacent workflow has the following characteristics:

- On each site *A*,*B*,*C* and *D*, different users manage large data sets;
- Data cannot be moved between sites due to issues related to data ownership, data privacy and data communication costs;
- Each user *uA*, *uB*, *uC* and *uD* knows about their data models and develops or uses software components with appropriate algorithms to process the local data sets;
- Users establish an international cooperation project to conduct multinational experiments, where sites *A* and *B* at North and South America continuously produce

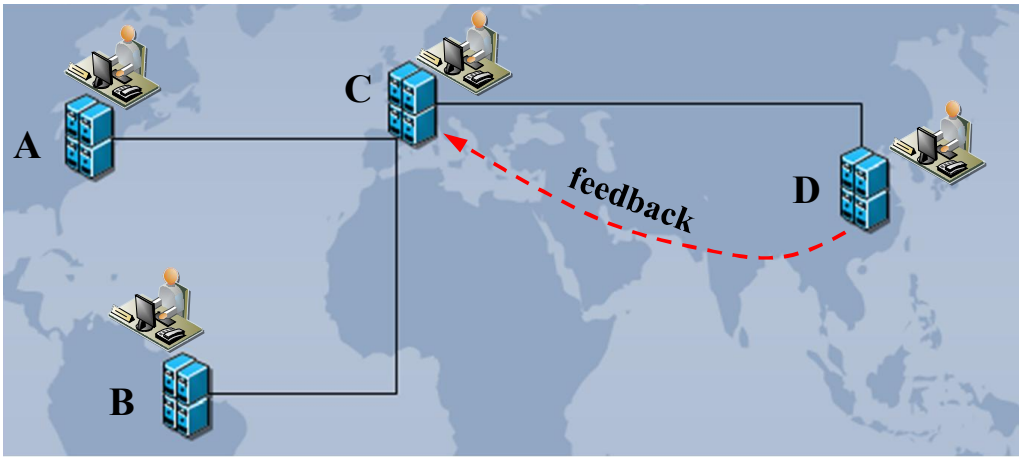


Figure 6.40: Scenario of a multinational scientific cooperation

small data sets with the results obtained from their local experiments, for instance, the main summary information related to sea tides and currents. Then site C uses the information received from sites A and B as inputs to new forecasting algorithms, while site D receives the results from the C site.

These continuous experiments generate long-running iterative computations, which for each step can be functionally defined by Equation 6.1, where ${f^i}x$ represents the algorithm functionality at site $x \in \{A, B, C, D\}$ at iteration i.

$$f experiment^i = {f^i}_D({f^i}_C({f^i}_A(), {f^i}_B())) \tag{6.1}$$

In order to improve the predictions, each user should be allowed to independently change their algorithms locally without the need to stop or restart the global experiment, thus provoking only side effects on the results supplied as inputs to the other sites along the experiment chain. For instance, at iteration n a user should be able to dynamically reconfigure the local algorithm ${f^n}x$ in order to enhance the results. Then on site C at iterations i and j with $i < j$ it is possible that ${f^i}_C \neq {f^j}_C$ meaning that the user has changed the corresponding algorithm. Furthermore, at certain iteration the users in sites C and D can agree to introduce improvements on their algorithms, using feedback information (Figure 6.40, the dashed arrow from D to C), meaning that the function in site C shall be replaced to receive one more argument with feedback information from the D site.

In this case and assuming this dynamic reconfiguration occurs at iteration k the functionality of the experiment at iteration $k+1$ can be described by Equation 6.2, where ${f^k}_{Dfeedback}()$ represents the feedback information used to improve the algorithm of the C site.

$$f experiment^{k+1} = {f^{k+1}}_D({f^{k+1}}_C({f^{k+1}}_A(), {f^{k+1}}_B(), {f^k}_{Dfeedback}))) \tag{6.2}$$

The main characteristics of this application case scenario are: i) The large data sets remain on their local sites and there are not large amounts of data moving between sites. Only small amounts of intermediate data are moved, typically to enable the next

activities on the experiment chain; ii) The local activities are independently launched and monitored by different users; iii) The experiment is a long-running workflow with an infinite number of iterations, where the activities can be executed at different paces; and iv) Users should be able to steering local activities by dynamic changes on algorithms and their *Parameters*.

✧ **Case contribution for the AWARD evaluation**

This application case allows the AWARD evaluation with the following contributions: i) Exploiting the flexibility of the AWARD framework to easily support dynamic reconfigurations in a common application scenario where distributed scientific experiments are performed by multiple interactive users; ii) Avoiding the need of restarting the execution of long-running workflows when structural and behavioral reconfigurations are needed; and iii) Enabling independent and autonomous modification of multiple application components and inclusion of feedback loops into an ongoing distributed scientific experiment.

The implemented scenario shows the AWARD feasibility to run workflow activities on distributed data centers in different countries without the need of moving large amounts of data. The scenario also shows the AWARD functionality to support workflow activities being independently monitored and dynamically reconfigured by different users. These reconfigurations allow changing the *Task* algorithms to improve the computation results or changing the workflow structure to support feedback dependencies where an activity receives feedback input from a downstream activity. We present experimental results of an implementation of one practical scenario running on multiple data centers of the Amazon cloud with steering by multiple users.

✧ **The workflow specification**

To demonstrate the execution of an application with similar requirements as the above scenario, we have used the workflow depicted in Figure 6.41, which performs an infinite number of iterations.

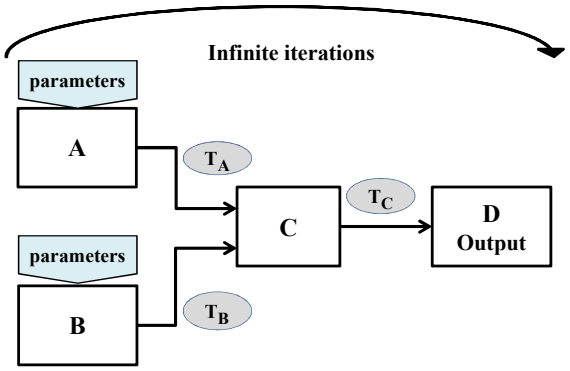


Figure 6.41: An evaluation workflow executed by different users

The *A*, *B* and *C* activities should be deployed and executed independently, for example on the Amazon cloud infrastructure. The *D* activity could be executed on a scientist

desktop machine in order to output the iterations results. For each iteration the *Tasks* of the *A*, *B* and *C* activities emit, respectively, the T_A, T_B, T_C tokens described respectively in Equations 6.3, 6.4 and 6.5:

$$T_A = (TaskA([parameter\ list]), \Delta_{tA}) \tag{6.3}$$

$$T_B = (TaskB([parameter\ list]), \Delta_{tB}) \tag{6.4}$$

$$T_C = (TaskC(T_A, T_B), \Delta_{tC})[\Delta_\varepsilon] \tag{6.5}$$

where Δ_{tA}, Δ_{tB} and Δ_{tC} are, respectively, the elapsed execution time of the *A*, *B* and *C* activities, and $\Delta_\varepsilon = \max(\Delta_{tA}, \Delta_{tB}) + \Delta_{tC}$ is the accumulated elapsed execution time until the completion of the *C* activity.

The *Task* of the *D* activity receives the T_C tokens emitted by the *C* activity, showing them through a graphical user interface component allowing a user to monitor the intermediate results for each workflow iteration.

Furthermore different users on different computers should be able to monitor and reconfigure other activities, for instance, by changing *Parameters* and the *Task* algorithms of the *A* and *B* activities.

To illustrate the case of feedback dependencies we submit a dynamic reconfiguration plan to the workflow in Figure 6.41 leading to the workflow depicted in Figure 6.42.

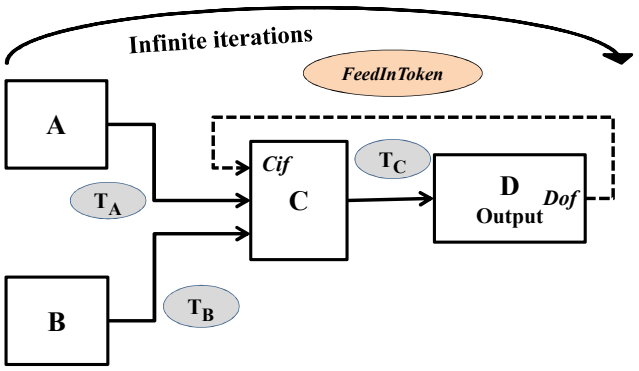


Figure 6.42: The workflow after a dynamic reconfiguration with a feedback loop

This scenario requires structural and behavioral workflow changes performed by the following outlined reconfiguration plan:

1. **CreateOutput** *Dof* on the *D* activity;
2. **ChangeTask** on the *D* activity to produce tokens to the *Dof* output port;
3. **CreateInput** *Cif* on the *C* activity;
4. **ChangeTask** on the *C* activity to receive one more argument. Then *TaskC* is changed to emits tokens $T_C = (TaskC(FeedInToken, T_A, T_B), \Delta_{tC})[\Delta_\varepsilon]$ where *FeedInToken* carries the date and time of the computer where the *D* activity runs.

✧ **Implementation**

In the following, we demonstrate how the requirements of the above scenario are met by an operational implementation under the AWARD framework and present experimental results as a proof of concept.

We developed the *Task* of the *A* activity, named *TaskA* to emit periodically, on each 2000 milliseconds the T_A token (as in Equation 6.3) and the *Task* of the *B* activity, named *TaskB* to emit periodically, on each 1000 milliseconds the T_B token (as in Equation 6.4). The *C* activity reads tokens from its two input ports and its *Task*, named *TaskC*, imposes a delay of 4000 milliseconds and then emits the T_C token (as in Equation 6.5) converted as a string.

Therefore the *D* activity displays the workflow execution results as shown in Figure 6.43 where we can observe that the accumulated elapsed execution time for each iteration is 6000 milliseconds.

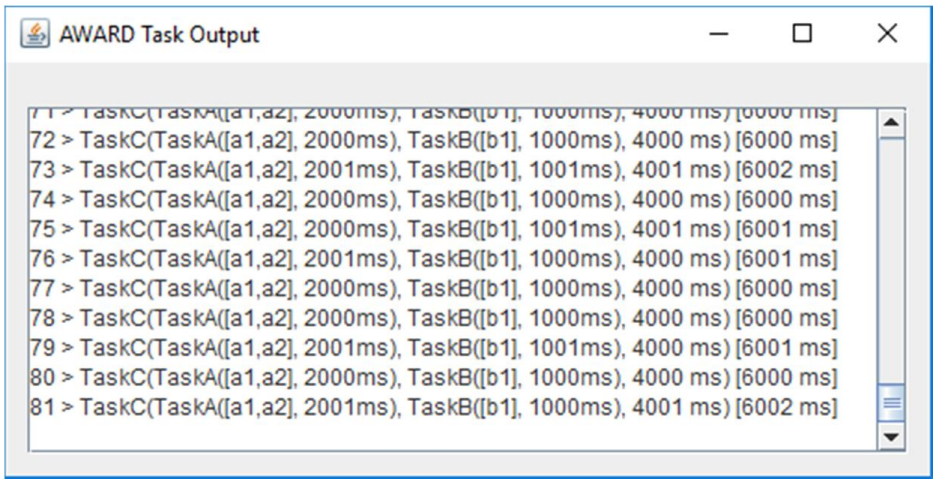


Figure 6.43: Activity *D* displays the workflow results

✧ **Feasibility**

The experiments used the following mappings for executing the workflows of Figure 6.41 and Figure 6.42:

- The *A* and *B* activities were launched on Amazon EC2 Linux virtual machines in US data centers;
- The *C* activity was launched on the Amazon EC2 Linux virtual machine in Ireland data center;
- The *D* (Output) activity is launched with a graphical interface (GUI) on a desktop computer in Lisboa allowing a user constantly monitoring the results as shown in Figure 6.43;
- The AWARD Space server was running in a Linux virtual machine hosted on Amazon Ireland data center.

Regardless of the location of the above Linux virtual machines, all of them have been instantiated from the same EC2 image, previously configured with the AWARD environment including the implementation of the activity *Tasks*.

Using the *AwardGUI* tool (see Section 5.9.2) it is easy to dynamically change the software component that implements the *Tasks* in the workflow activities by submitting reconfiguration plans. In Figure 6.44 we show the use case to change the *Task* of the *C* activity to the new *TaskCnew Task* that occurred at the 41st iteration.

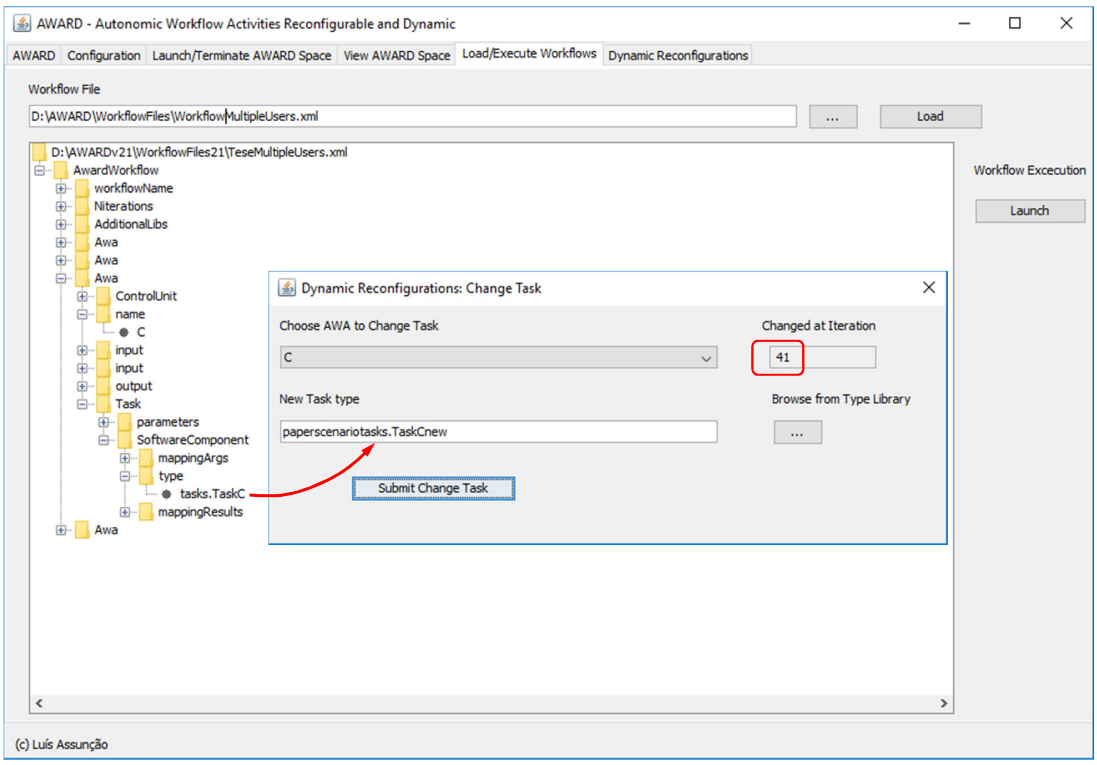


Figure 6.44: The AWARD tool for changing the *Task* of the *C* activity

Figure 6.45 presents the graphical user interface of the *D* (Output) activity where we can see that after the 37th iteration the user of the *B* activity changed the *b1* value of the *TaskB* parameter to a new *b1new* value and at the 41st iteration the user of the *C* activity changed to a new *TaskCnew Task*.

The new *Task* of the *C* activity decreased from 4000 milliseconds to 2000 milliseconds, illustrating how algorithm improvements can be achieved by dynamic reconfigurations.

To evaluate the workflow scenario of Figure 6.42 on page 253, we submitted the reconfiguration plan presented in Listing 6.16 involving the *C* and *D* activities.

The reconfiguration plan of Listing 6.16 that led to the workflow illustrated in Figure 6.42 on page 253 changes the *Task* of the *D* activity to emit tokens with the current time on the new *Dof* output port connected to the new *Cif* input port of the *C* activity. Thus the new *Task* of the *C* activity, named *TaskCwithFeedback* receives three arguments where the first one is the feedback token sent by the *D* activity.

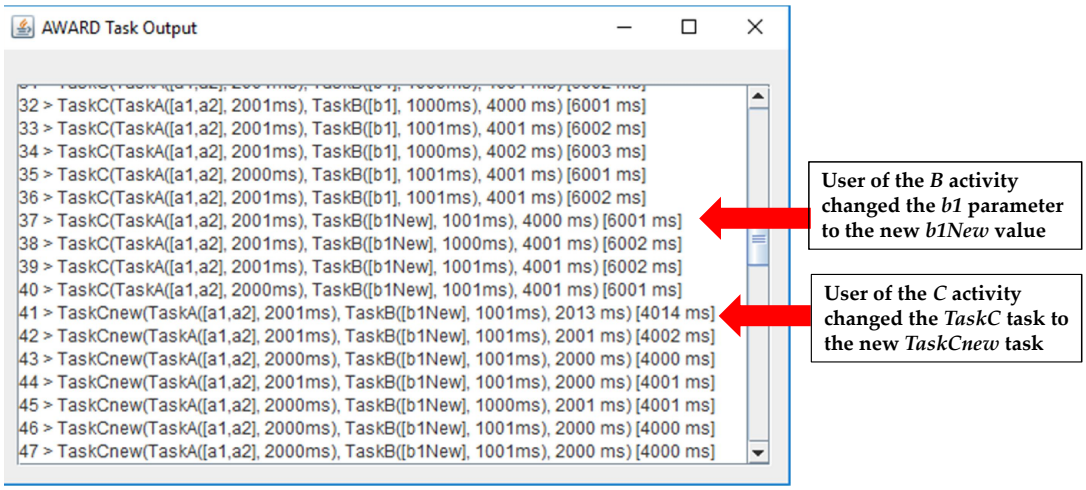


Figure 6.45: Visualizing on activity *D* the effects of dynamic reconfigurations

Listing 6.16: Reconfiguration plan for introducing feedback between activities *D* and *C*

```
int RID = BeginReConfiguration(new String[]{"D", "C"});
  BeginAwaReConfig(RID, "D");
    CreateOutput(RID, "D", "Dof", "EnableFeedback", "java.lang.String", "Single");
    AddOutputLink(RID, "D", "Dof", new String[]{"Cif"});
    ChangeMappingOutputs(RID, "D", new String[]{"Dof"});
    ChangeTask(RID, "D", "tasks.TaskDwithOutput");
  int it1 = EndAwaReConfig(RID, "D");
  BeginAwaReConfig(RID, "C");
    CreateInput(RID, "C", "cif", "EnableFeedback", "java.lang.String", "Iteration");
    ChangeTask(RID, "C", "tasks.TaskCwithFeedback");
    ChangeMappingInputs(RID, "C", new String[]{"cif", "ci1", "ci2"});
  int it2 = EndAwaReConfig(RID, "C");
  int[] AgreementSet=new int[]{it1, it2};
int K=EndReConfiguration(RID, new String[]{"D", "C"}, AgreementSet);
```

The result of applying the reconfiguration plan with an agreement at the 23rd iteration is shown in Figure 6.46. The reconfiguration plan changed the *Task* of the *D* activity for sending feedback to the *C* activity processed at the 24th iteration.

✧ **Case conclusion**

Although, at first sight, there is an apparent simplicity in these functionalities, the execution of this workflow enables the following main characteristics:

- The execution is distributed by different data centers;
- Each activity is launched, monitored and dynamically reconfigured by different users;
- The operation of the workflow activities proceeds in a completely decentralized way, under the AWARD framework.

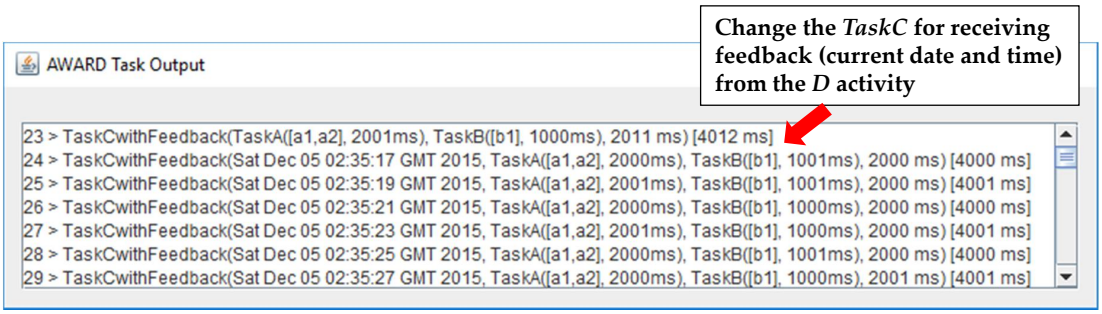


Figure 6.46: The *D* activity shows the effect of feedback between the *D* and *C* activities

6.4.5 Case 5: Text Mining Application

✧ Description

Applications for information retrieval and data analytics require automatic mechanisms to select and extract relevant information from large-scale document collections. The *LocalMaxs* algorithm [Sil+99] is based on a statistical language-independent approach used to extract relevant expressions from text *corpora*. Relevant expressions are groups of words (n-grams) in a *corpus* that are recognized as lexical units with strong semantic meaning, such as *"parallel processing"*, or *"global finance crisis"*. In this method, the extraction of relevant expressions is based on the counting of the n-gram occurrences, and on the evaluation of a metric that reflect the cohesion or "glue" of the identified n-grams, followed by the decision on the relevance of the extracted n-grams. The algorithm consists of a sequence of three phases that are modeled by a pipeline as illustrated in Figure 6.47.

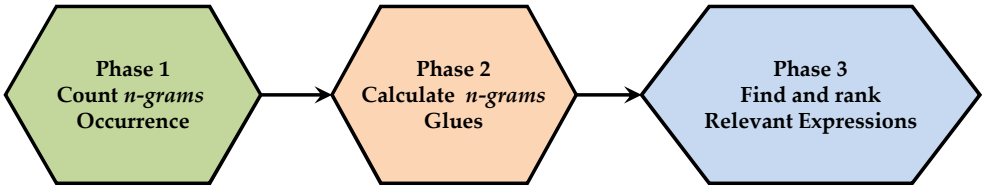


Figure 6.47: Pipeline of the *LocalMaxs* algorithm

The *LocalMaxs* algorithm was proposed by [Sil+99] and its parallel and distributed implementation was developed in an ongoing research project [SC15] in the context of the PhD dissertation by Carlos Gonçalves [Gon17]. The description of the parallel version of the *LocalMaxs* is summarized here and illustrated as a workflow in Figure 6.47 and Figure 6.48, was developed by Carlos Gonçalves [GSC15].

The entire development of the parallel *LocalMaxs* application workflow was made using the AWARD model and environment as a basis, in order to specify the *LocalMaxs Task* dependencies, their alternatives for parallelization at each phase, and also to manage the mappings of the workflow activities onto virtual machine nodes, in cluster and cloud environments. The mechanisms for monitoring and debugging, as provided by the AWARD environment, have also been intensively used for supporting the parallel

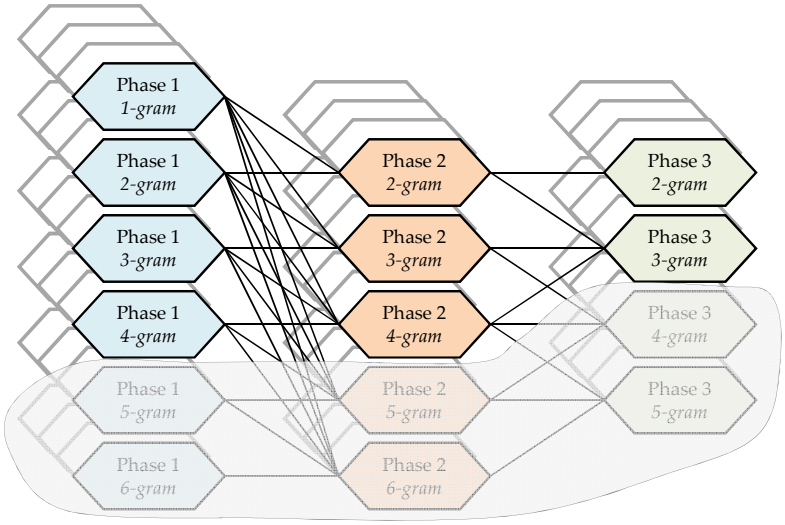


Figure 6.48: Workflow for parallelizing the *LocalMaxs* algorithm

LocalMaxs implementation.

The fact that the AWARD model, its runtime environment, and its support tools, have been intensively used by an external independent user, that is, someone who was not involved neither in the design nor in the implementation of the AWARD model, represents a clear indication, although still preliminary, that the AWARD framework, as proposed in this dissertation, provides the required functionality for expressing real robust applications, and is feasible to support their development and execution, by other users.

In the above mentioned project, the automatic extraction of relevant expressions from large text *corpora* was developed as workflow structures with multiple nodes for processing the input *corpus* until 4-gram on phase 1, processing glues until 4 gram in phase 2 and processing the relevant expression with 2-gram and 3-gram on phase 3 as illustrated in Figure 6.48 [GSC15].

In the experiments for supporting large *corpus* AWARD workflows have been used where each activity is replicated in parallel up to 18 instances making workflows up to 162 activities for running up to 54 virtual machines hosted on cloud infrastructures.

✧ **Case contribution for the AWARD evaluation**

The AWARD flexibility for specifying workflows for parallel and distributed execution of multiple activities, the independence of the *AwardExecutor versus* the activities *Task* internals and the AWARD feasibility for executing workflows on cluster and cloud infrastructures with dozens of virtual machines has been a key f*actor* for effectively supporting the intensive experimentation involving large text *corpus* [GSC15].

✧ **Case conclusion**

This application case is very important for demonstrating the following AWARD characteristics:

1. An independent developer that only knows the syntax and the programmer's view for specifying AWARD workflows as well as the AWARD tools, namely how to start the AWARD Space and how to launch workflow activities, was able to use the AWARD framework in order to develop and run effective working application workflows without knowledge on the AWARD machine implementation;

2. The functionality of the AWARD environment for running distributed workflows following application-dependent strategies by spreading multiple activities (up to 162) on multiple computing nodes (up to 54) of cluster or cloud platforms;

3. The total freedom for an application developer to program AWARD activity *Tasks* as complex algorithms involving multithreading and in-memory data storage, totally decoupled from the internals of the AWARD machine.

6.5 Comparison of AWARD with other Workflow Systems

In the beginning of this dissertation work and even during the development of the AWARD framework we found a plethora of initiatives related to scientific workflows. However, many of these initiatives did not provide reusable workflow systems and tools with easily installation and configuration in order to be used for developing workflow applications. Therefore, in this section, the AWARD framework is compared with other workflow systems regarding the support of the following dimensions:

1. Workflow execution on heterogeneous computing environments, for instance on a simple standalone computer as well as on distributed infrastructures;

2. Dynamic reconfiguration of long-running workflows;

3. Decentralized control for executing the workflow activities with autonomic characteristics.

✧ **Parallel and distributed execution**

The support for parallel and distributed workflow execution decoupled from any specific middleware was one of the main motivations for developing the AWARD model. The AWARD implementation only depends on Java technologies so the execution of AWARD workflows is possible practically on all computing environments.

Typically some of the existing workflow systems are tightly coupled with specific middleware technologies or even with particular versions of operating systems. For example, the Pegasus workflow management system [Dee+15] that has been widely used to map abstract workflows into concrete execution plans, has dependencies on specific middleware for executing the workflow activities as jobs on distributed platforms, such as Condor or Globus. Pegasus is not supported in the Windows operating system, its model only supports direct acyclic graph (DAG) workflows and does not support neither iterations nor feedback loops. The Askalon [WPF05] workflow system developed the

Abstract Grid Workflow Language (AGWL) for describing grid workflow applications at a high level of abstraction but a centralized workflow execution engine is required to interpret the AGWL workflow and to map the jobs to grid resources managed by low-level scheduling services such as PBS [HT96] and HTCondor [TTM05] wrapped by the Globus middleware.

The success of service oriented architectures based on Web Service standards has driven organizations to developing specialized Web Services for solving problems or processing data in several science domains. For example, Taverna [Wol+13] was developed for combining distributed Web Services and local tools into workflow pipelines that can be executed on desktop computers or through distributed infrastructures, in the grid or the cloud. The Taverna workbench allows users to select and combine Web Services by dragging and dropping them onto the workflow design panel. The Taverna server provides workflows from bioinformatics and biodiversity area. However, the approach for developing workflows based on third party resources, for instance Web Services, originates issues related to the volatility of the resources required for workflow executions, which can provoke breaks or unpredictable results on workflow rerunning [Zha+12].

Although the user interface for designing workflow graphs was not a topic addressed in this dissertation it is an important practical issue mainly for workflows with a large number of activities.

Triana and Kepler systems are widely used given their characteristics of having graphical user friendly interfaces for visual designing of workflow graphs and are developed in Java as open-source projects so they are easily installed on any computer. Both allow designing of workflows by drag and drop pre-existing workflow activities as software components called *units* in Triana and *actors* in Kepler.

Triana also incorporates modules, such as *Grid Application Prototype* (GAP) and *Grid Application Toolkit* (GAT) [Chu+06; Tay+03] for integrating existing Grid Services and Web Services as well as peer-to-peer communication for allowing remote service invocations. Distributed execution of Triana workflows relies on a specialized *unit* for distributing any task or group of tasks as subworkflows. However, the Triana core processing *units* as well as *units* for data transformation and visualization are executed by a centralized execution engine called *Triana Controlling Service*.

In Kepler [God+09] the basic workflow components are *directors* and *actors*. Following one model of computation a workflow *director* controls the workflow execution and *actors* perform the workflow computations/tasks by taking instructions from the *director*. The communication between *actors* for sending data or message tokens is performed through input and output ports. Each *actor* sends tokens to an *actor* connected to one of output ports. An *actor* runs the required number of iterations where after receiving tokens on its input ports it fires and generates new tokens with the resulting data on the output ports. *Actors* can be grouped into a *composite actor* as a set of *actors* bundled together in order to perform more complex operations used in workflows as nested subworkflows. A *composite actor* has its own *director*, which can be different from the *director* used in the

parent workflow. Distributed workflow execution is achieved by using *composite actors* and by extension of *directors* for transporting and executing workflows and subworkflows across a distributed set of computing nodes in order to improve execution performance. In [Plo+13] some distributed execution techniques are presented for Kepler workflows including a distributed data parallel framework using Hadoop [Apa15a] and grid execution by using specific *actors*.

Furthermore and unfortunately the above described Triana and Kepler functionalities for distributed execution of workflows are not yet available as of December 2015 in the downloadable system versions. Therefore the feasibility for end users taking advantages of these functionalities is not easy. This is more noticeable in Triana version 4 [Tri15].

One of the motivations for developing the AWARD model and its associated working operational runtime environment aimed at providing flexibility and feasibility to enable the practical execution of the workflow activities on distributed infrastructures.

When comparing AWARD with the above systems there is a significant difference. Workflow activities in AWARD are autonomic and can be separately executed and controlled by different users on distributed computing nodes without any centralized control. In the above systems some form of centralized control remains. For instance both Triana and Kepler have centralized control for executing the global workflow where special *units* or *actors* can encapsulate distributed executions of subworkflows or invoking remote services. These Triana *units* or Kepler *actors* functionalities can easily be implemented inside AWARD activities. In fact, an AWARD activity *Task* can be transparently programmed for using any Java grid API.

In terms of non-basic workflow patterns Triana supports loops and execution branching handled by specific *units* with semantics not easy to use.

Kepler supports feedback loops using the built-in *SampleDelay actor* as illustrated in the workflow of Figure 6.49.

The workflow is executed for 10 iterations and for each iteration the *Ramp actor* generates integer values from 0 until 9 to be added by the *Add actor*. For the first iteration the *SampleDelay actor* fires a default value that is configured to zero and for the next iterations the *SampleDelay actor* carries its input port value (the result of the previous iteration plus 1) to its output port that connects to the *Add actor*. The results of the 10 iterations are shown by the *Output actor*.

AWARD supports feedback loops with a clear and useful semantics as presented in Section 6.3.3, including the introduction of feedback loops by using dynamic reconfiguration plans. Kepler only supports feedback loops at workflow design time because it is not possible to dynamically change the workflow structure during the workflow execution.

Furthermore both Triana and Kepler do not support the AWARD load balancing pattern.

Kepler has a rich set of *directors* for supporting a flexible set of models of computation [God+09] including *Synchronous Data Flow* (SDF), *Continuous Time* (CT), *Dynamic Data Flow* (DDF), *Process Network* (PN) and others.

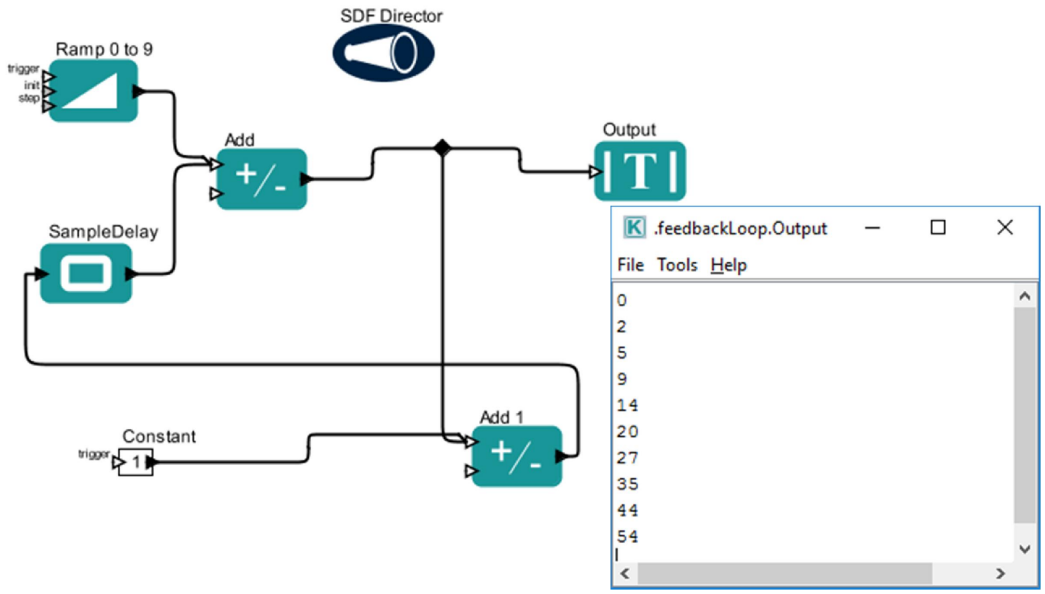


Figure 6.49: Workflow with a feedback loop in Kepler

There are important differences between Kepler and AWARD.

In Kepler the parallelism is based on executing each *actor* by threads within the same monolithic process, and *actors* communicate using first-in-first-out memory buffers. The execution order (*actors* firing) is controlled by a centralized PN Director that can be very inefficient, as it must keep looking for *actors* with sufficient data to fire. If one *actor* fires at a much higher rate than another, the *actors'* memory buffers may overflow, causing workflow execution to fail [Kep14]. In addition the PN Director does not manage iterations, possibly leading to non-determinism and undefined termination of the execution. For instance, in *Composite actors* with workflow hierarchies, if two *actors* have computation threads, it is ambiguous which *actor* should be allowed to perform computation [God+09]. Instead, AWARD activities are encapsulated in parallel processes (AWA) and can execute in distributed environments without a centralized control. Each AWA activity has autonomic control and communicates through the AWARD Space by producing/consuming tokens at different rates without overflow problems. AWARD supports the notion of iterations allowing determinism and well-defined termination of the AWA activities.

The current implementation of the AWARD Space uses a shared tuple space. Using tuples for token communication improves flexibility by supporting different granularities in complex data types and easily enabling the AWARD support for dynamic workflow reconfigurations.

Although with different objectives other works also rely on tuple spaces. For example the *Workflow Enactment Engine* (WFEE) [YB04] uses a tuple space for event-based notification for just-in-time scheduling. In [HPA05] tuple spaces are mainly used to coordinate request and notification events between *Navigators* (workflow engines) and the *Dispatchers* (Task executors). However, none of these works use tuple spaces to support

the Process Networks (PN) model.

Despite the advantages of the tuple spaces model it also raises difficulties related to scalability and bottlenecks [OG02]. To address these problems some works rely on distributed tuple spaces, using caching and replication techniques. For instance, Comet, a decentralized tuple space [LP05] and Rudder [LP06; LP07], which provides a software agents framework for dynamic discovery of services, enactment and management of workflows, where an interaction space is used to coordinate task scheduling among a set of workers. Similar to the AWARD space, the task tuples contain data items among workflow nodes.

As the AWARD model is orthogonal to the tuple space implementation, we argue that it is possible to map the AWARD Space onto distributed tuple spaces, such as the Comet space [LP05] or the Tupleware [Atk08].

Closer to AWARD, [AB11] proposes a framework based on persistent queues to support flow between activities, but with a monolithic workflow execution engine. Tasks are also executed as threads, not allowing the execution of workflow activities on distributed infrastructures. Additionally, [AB11] claims that persistent queues can easily support provenance storage. This claim also applies to AWARD, as far as tuple spaces, including IBM TSpaces, also support persistence. In [FTP11] a workflow system is presented with a decentralized architecture with an external storage (*Multiset*) as a shared space between workflow activities encapsulating a chemical engine. Unlike AWARD Space, the *Multiset* contains coordination information and the workflow definition.

Although the goal to support flexibility in business workflows has been a concern since the nineties [DR09] many issues still remain open, regarding the support for dynamic changes [BL10]. Such issues are also important in scientific workflows, and some are supported by AWARD. Namely, supporting dynamic behavioral changes, for example, by changing the execution $Task$ and its $Parameters$ at runtime it is important in scientific experiments where the behavior of algorithms and their $Parameters$ are not known in advance. A Kepler workflow [Kep14] is static and must be completely specified before starting the execution, not allowing changes during run-time. However, a prototype implementation based on Kepler [Ngu+08], proposes a frame abstraction as a placeholder for *actors* to be instantiated at runtime (dynamic embedding), according to rules defined at design time. This approach can be compared to our proposal to change dynamically the algorithm of an activity. However, AWARD is more flexible because we do not need to specify the alternative algorithms at design time as we allow to dynamically changing them by invoking dynamic operators to inject tuples with the new tasks. In the GridBus workflow execution engine [YB10] the user can either specify the location of a particular service at design time, or leave it open until the execution engine identifies service providers at run-time. This is easily supported in AWARD. If the location of the service (URL) is a parameter we can dynamically reconfigure the parameter to change the service provider, or we can inject rules into the control unit of an AWA activity in order to search for service providers in any service directory. Other approaches exploit the

runtime reconfiguration of distributed services for managing the application behavior [Tsa+04; Vaq+12] but they are not directly focused on workflow reconfigurations.

✧ **Dynamic reconfigurations**

The need to provide adequate solutions to handle the dynamic modification in application and infrastructure behavior has motivated research efforts in dynamic reconfigurable workflow models [Gil+07; RRD04].

Faults and other kinds of situations originating expected and unexpected events at the workflow, middleware and resource infrastructure levels are usually handled by separate mechanisms by the workflow execution engine or runtime environment [Gil+07; HA00; HK03; Lac+10].

Most of the existing approaches only address very specific concerns involving a limited level of dynamic reconfiguration of the workflows, usually for restricted scenarios and well-known and expected situations whose handling is predefined at workflow design time. For example, in some of the more popular scientific workflow tools such as Taverna [Tav11], Triana [Tri11], or Kepler [Kep13], although there is some support for user-handling of faults, the allowed recovery actions follow strategies such as retrying workflow tasks a certain number of times, or replacing them at runtime, by considering the selection of alternative handling candidates defined at development time, only applying them to well-defined and expected event types.

Other approaches are also restricted to predefined strategies at workflow design time using for example workflow patterns [RAH06; TC+10] for dynamic replacement of subworkflows on the occurrence of exceptions.

However, on existing workflow systems, for example Kepler, there is still a lack of general-purpose mechanisms and more unified approaches for supporting dynamic reconfiguration behaviors within the scope of current workflow framework and tools.

The AWARD support for dynamic reconfigurations is, to the best of our knowledge, a distinctive characteristic as demonstrated in several evaluation scenarios and application cases, where failures and degradation of quality of service situations are addressed by dynamic reconfigurations, allowing to handle unexpected event through user intervention, and also allowing automated handled of well-identified situations. This is achieved by relying on the mechanisms provided by the AWARD framework.

✧ **The autonomic characteristics of the AWARD model**

In the following we discuss why we consider the AWARD workflow activities (AWA) as autonomic components.

An autonomic computing system consists of several autonomic elements, each of them managing its internal behavior and relationships to other autonomic elements based in policies specified by users or induced by the execution environment. Any autonomic component needs an associated manager [KC03], as in Figure 6.50. Through a set of sensors the manager constantly monitors and collects events originated internally or in the surrounding environment. These events are analyzed and a changing plan enforces

the current policies by provoking the execution of self-changes involving the autonomic component.

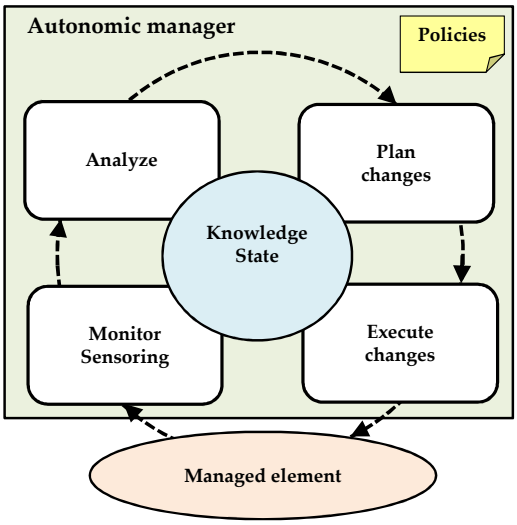


Figure 6.50: The architecture of an autonomic component [KC03]

Any AWA activity acts as an autonomic component because the AWARD *Autonomic Controller* supports the following main capabilities: i) The knowledge state is the working memory of the *Rules Engine* where facts and rules are stored for defining the states in the life-cycle of an AWA activity; ii) The asynchronous event handlers, such as the *Dynamic Reconfiguration Handler* act as sensors to detect special events in the AWARD Space, for supporting dynamic self-reconfigurations; iii) Sequences of these events into the AWARD Space allow preparing a plan to change the structure or behavior of the AWA activity. This plan consists of a set of new facts and rules in the working memory of the *Rules Engine* which provokes a self-reconfiguration by changing the *AWA Context* when an agreed upon iteration is reached and the *State Machine* executes the change plans in the *Config* state, as presented in Chapter 4.

Thus, AWARD workflows have autonomic attributes driven by the self-behaviors of each AWA activity, or driven by external tools as interpreters of the reconfiguration plans using the *DynamicLibrary* that inject the adequate events into the AWARD Space to produce workflow changes.

For many useful scenarios the consistency of dynamic reconfiguration (Definition 4.5 on page 94) is achieved by relying on a global iteration agreement, between all activities involved (Definition 4.6 on page 98), associated with the specification of scripts for performing reconfiguration plans.

Nowadays a big challenge is how to unify a large number of distributed autonomic elements into a global autonomic system [Kep05]. The AWARD framework deals with this challenge by supporting dynamic reconfigurations plans that affect large number of distributed AWA activities.

6.6 Chapter Conclusions

The evaluation of the AWARD model and its implementation was performed by using significant scenarios for allowing practical experiments in three dimensions:

1. **Parallel and distributed workflows execution:** We demonstrate the AWARD functionality and expressiveness for supporting workflows with basic patterns and the feasibility for executing them on distributed computing infrastructures namely on the Amazon cloud. Although it is a topic that is out of the scope of our work we also discuss some performance indicators when we spread the workflow activities to multiple computing nodes taking advantage of the decentralized control of the workflow activities;

2. **Structural and behavioral dynamic reconfigurations:** We demonstrate the more distinctive AWARD characteristic to support dynamic workflow reconfiguration plans expressed as sequences of operators provided by an operational Java software library. This support was discussed by experimenting with useful scenarios where long-running workflows are reconfigured with clear advantages;

3. **Functionality, expressiveness and feasibility to develop real application cases:** We illustrate the practical use of the model in real application scenarios: i) A concrete workflow for implementing the MapReduce model; ii) Workflows whose activities invoke third party Web Services; iii) The support for fault recovery by taking advantage of dynamic reconfigurations; iv) Workflow steering and reconfiguration by multiple users; and v) A text mining application developed by an independent user which includes lauching multiple activities executed on a local cluster and on two public cloud infrastructures (Amazon and LunaCloud).

The extensive experimentation with useful and concrete application cases and the results achieved allow us to conclude that the AWARD model and its implementation is suitable to be used for application development using the scientific workflows paradigm.

Chapter 7

Conclusions

Dissertation conclusions and directions for future work.

This chapter presents the dissertation conclusions and identifies possible directions for future work.

In Section 7.1 we outline the dissertation context and the dimensions related to the AWARD (Autonomic Workflow Activities Reconfigurable and Dynamic) workflow model.

In Section 7.2, we summarize the relevant characteristics of the AWARD model and the contributions of this dissertation. We also mention the publications in the context of this dissertation.

In Section 7.3 we discuss the contributions and lessons learned concerning the AWARD model and its implementation.

In Section 7.4 we discuss several relevant open issues and identify directions for future work.

7.1 Outline of Dissertation Dimensions

The scientific workflow paradigm has been used for developing complex applications based on problem decomposition into multiple activities. Workflows also facilitate large scientific experiments where different users with expertise on distinct scientific domains may develop specific activities that can be combined to execute applications in the available parallel and distributed computing environments.

As also realized by the scientific community [Chi+11; Dee07; Dee+08] we found a need for improving the support provided by existing workflow tools [AGC09] in order to find solutions to several open issues.

Initially we studied the possibility for improving some of the existing systems, assuming they were based on available and well documented open-source code. However, we found great difficulties. On one hand, the existing systems were based on large amounts of source code that were not well organized and easy to reuse and, even worse, some code versions were changed almost daily. For example, the revisions of source code performed from Kepler 1.0 to Kepler 2.0 have increased the complexity for developing and integrating new components, including incompatibilities with components developed in the previous versions. On the other hand, the available workflow systems, for instance Triana and Kepler, were based on execution engines with centralized control, therefore requiring a great software engineering effort for supporting the approaches needed for solving the identified open issues.

Therefore we decided to develop and implement a new workflow model called AWARD (*Autonomic Workflow Activities Reconfigurable and Dynamic*), to address the identified open issues and to enable the development of scientific workflows with support for structural and behavioral dynamic reconfigurations.

This dissertation aims at contributing to increase the flexibility and efficiency of the problem solving process using scientific workflows by taking advantage of the AWARD workflow model. An important goal of this dissertation was the implementation of a working prototype, allowing experimentation and enabling continuous refinements to the AWARD model in order to support the feasible execution of the concrete applications. This goal required a great effort to provide a flexible and transparent implementation of an AWARD machine decoupled from disparate technologies and capable of being reused in multiple computing environments. The AWARD machine implementation, the software library for supporting dynamic workflow reconfiguration plans and the AWARD tools for launching and monitoring the workflow execution, they together provide an operational framework for executing workflows in distinct computational environments exploring forms of parallelism and distribution currently available on the cluster and cloud computational infrastructures.

In this dissertation the AWARD framework was evaluated in multiple scenarios. Besides a set of workflow template scenarios, which were used to exercise and evaluate the AWARD model characteristics and their implementation, the use of the AWARD framework was also evaluated through a set of concrete application cases where the main concerns of expressing parallelism and distribution, performing dynamic reconfigurations and supporting real experimentation on distributed computing infrastructures, were assessed. The effectiveness of the AWARD model and its prototype implementation were also evaluated by an external user that developed a text mining application, and only needed to know the programmer's view for specifying AWARD workflows and the associated AWARD tools for launching and monitoring the workflow execution in distinct computational infrastructures.

7.2 Characteristics and Contributions of the AWARD Model

In the following the AWARD model characteristics are summarized according to several dimensions related to the model itself, the operation of the AWARD machine, the implementation of the AWARD framework, and the experimental evaluation for workflow development including several international publications that also contributed to the validation of the results achieved by this dissertation.

7.2.1 The Model

The AWARD model provides adequate expressiveness and flexibility to support multiple workflow patterns allowing application parallel decomposition. In addition to the support of basic workflow patterns, the AWARD model also contributes with support to non-basic patterns, such as feedback loops between workflow activities and replication of activities executed in parallel for load balancing purposes.

The AWARD model offers several types of transparency. The token types used for passing information between workflow activities are application-dependent and totally decoupled from the execution engine. The *Task* development of the workflow activities is decoupled from the underlying execution infrastructure. In fact, a *Task* is any Java class with a generic well-defined entry point for receiving a collection of objects mapped from the tokens available at the activity input ports, and on its completion the *Task* returns an object collection to be mapped to the activity output ports. Furthermore, the AWARD workflow specification is decoupled from the mappings to the execution environments.

The same workflow specification is executable on multiple standalone computers on a local network, on multiple nodes of a cluster, or on clouds using multiple virtual machines by only requiring small adjustments to the configuration of the AWARD execution environment (Listing 5.10 on page 168). Due to such transparency, the workflow developers can focus on the problem domain and not on the intricacies of the implementation of the workflow execution engine.

The AWARD model provides a set of reconfiguration operators for dynamically changing the workflow structure and behavior during the execution of long-running workflows. This is an innovative characteristic of the AWARD model which contributes to enable useful scenarios where workflows can be repeatedly adjusted by reconfiguration plans in order to dynamically achieve the application objectives. These scenarios include the behavior improvement of some activities by changing their *Tasks* and *Parameters*, for instance, for failure recovery, or even the modification of the workflow structure by introducing new activities, for instance to support feedback loop and load balancing patterns.

7.2.2 The Operational View of the AWARD Machine

The design of the AWARD machine is based on a decentralized execution control model where each *Autonomic Workflow Activity* (AWA) has an autonomic behavior and runs

without dependencies on a centralized execution engine. The AWA workflow activities execute multiple or even infinite number of iterations as a long-running workflow, where different activities can proceed asynchronously in different iterations. Each AWA activity consumes and produces tokens at its own pace and can terminate independently of the others.

The *Autonomic Controller* that executes an AWA activity has an internal architecture based on a *State Machine* and a *Rules Engine*, which provides openness to further extensions, for instance, to modify the life-cycle or the behavior of a particular activity.

The links between AWA activities are abstractions supported by the AWARD Space as a unbounded and reliable global data store. The AWARD Space is also used as intermediary between the external tools and the AWARD machine to coordinate the actions required for applying dynamic workflow reconfiguration plans during the workflow execution, involving one or more AWA activities.

These AWARD machine characteristics contribute to separate the workflow specification from the execution environment, by allowing the workflow activities to be launched and run separately on heterogeneous infrastructures, ranging from a single computer to distributed infrastructures, such as network of local computers, clusters and clouds. Depending on the specific application scenarios a workflow can be subdivided into partitions of multiple activities. Each partition can be separately launched on different distributed sites and monitored by different users.

7.2.3 The AWARD Framework

The AWARD framework is supported by an effective working prototype composed of the following components: The kernel for controlling the life-cycle of an AWA activity; the AWARD Space server; the dynamic software library that encapsulates the interface for applying dynamic reconfiguration plans; and a set of tools supporting the development and execution of AWARD workflows.

The AWARD framework was implemented using the Java language and has minimum dependencies upon third-party software components. In fact, the AWARD framework only depends on the Java JESS library to implement the *Rules Engine* and the Java *IBM TSpaces* library to implement the AWARD Space server.

The set of AWARD tools supports the life-cycle of workflow development. The associated tools are used for specifying workflows, mapping their execution to computing infrastructures, and monitoring and debugging them by analyzing the execution logs. There are tools to setup and to launch one or more activities on computing nodes, tools to force the termination of one or more activities and tools to manage the log information, for instance to inspect the activity execution times.

The AWARD tools and the dynamic reconfiguration library are very lightweight and easily portable to different computing environments. As two examples, the Java executable for executing an AWA activity, which implements the AWARD machine including

the necessary libraries, has a size less than 7 MB, and the AWARD Space, as a standalone Java application server, has a size less than 1 MB. A useful characteristic of the AWARD framework is to provide tools to inspect the workflow execution logs. For long-running workflows the observation of this log information is a crucial point to monitor the behavior of all workflow activities even those that are running on distinct computing nodes. In addition the log information is also very helpful for code debugging purposes. This was very important during the prototype implementation process but it was also shown important when used by workflow developers for performing application-level debugging. The AWARD tools provide great flexibility for running AWARD workflows on distinct types of computing nodes, such as, standalone computers as well as virtual machines on cluster and cloud infrastructures.

As was demonstrated and validated by multiple experimental scenarios the AWARD framework can easily be reused in multiple computing environments contributing to simplify the development of real application scenarios.

7.2.4 Evaluation of the Experimental Results

This dissertation proposes a flexible and user-friendly workflow model addressing several important requirements for developing scientific workflows. The dissertation also led to the development of the AWARD framework with a flexible and transparent architecture, whose implementation allows to explore the parallelism and distribution, currently available on cluster or cloud infrastructures.

The experiments aimed at evaluating how the AWARD model and the implementation of the AWARD framework are suitable to modeling multiple workflow scenarios and developing application cases in the following dimensions:

- **Parallel and distributed workflows execution:** We demonstrate the AWARD functionality and expressiveness for supporting workflows with basic patterns and executing them on distributed computing infrastructures namely on the Amazon cloud. We also discuss some performance indicators when the workflow activities are spread out on multiple computing nodes, by taking advantage of the decentralized execution control. We observed that for long-running workflows with thousands of iterations AWARD exhibited small overheads when compared with the Kepler workflow system, which uses a centralized execution engine;

- **Structural and behavioral dynamic reconfigurations:** We demonstrate a distinctive characteristic of the AWARD model to support dynamic workflow reconfiguration plans expressed as sequences of dynamically invoked operations. This contributes to enabling useful scenarios, such as recovering from failures by changing the activity *Tasks* or their *Parameters*, introducing activity replicas for load balancing purposes, and workflow steering by multiple users, where long-running workflows were reconfigured with clear advantages;

- **Developing real application cases:** We demonstrate the feasibility of using AWARD to develop real application cases, such as: i) A concrete workflow for implementing the MapReduce model; ii) Workflows whose activities invoke third-party Web Services; iii) Supporting fault recovery by taking advantage of dynamic reconfigurations; iv) Workflow steering and reconfiguration by multiple users; and v) A text mining application developed by an independent user, including activities executed on computing nodes in a local cluster and in two public cloud infrastructures (Amazon and LunaCloud).

All the above experiments demonstrated that the AWARD model and its implementation are adequate to execute and dynamically reconfigure long-running workflows on parallel and distributed infrastructures.

7.2.5 Publications

The main contributions related to the AWARD model were peer reviewed, presented and published in the proceedings of international scientific conferences [AGC09], [AGC12], [GAC12], [AC13] and [AC14]. Two of these conference publications originated invitations to extended versions published in scientific journals [GAC13] and [AGC14].

7.3 Discussion and Lessons Learned

Despite the intensive work in scientific workflows during the last decade, a large number of issues still remain unsolved. This dissertation proposes feasible solutions to some of these issues by providing the AWARD model and its implementation as the AWARD framework.

The following distinctive characteristics are highlighted: i) Expressiveness to support non-basic useful workflow patterns, such as feedback loops and load balancing; ii) Transparency to decouple the workflow specification from the execution engine and the underlying execution environments; iii) Support to apply dynamic reconfigurations to long-running workflows; and iv) Support to a decentralized control model allowing flexibility to execute workflows on parallel and distributed infrastructures.

Among other characteristics, it is important to distinguish the flexibility, transparency and ease of use of the AWARD framework for developing scientific workflows. This was effectively confirmed by an external user that only needed to know the AWARD programmer's view in order to develop a complex text mining application using parallelism and distribution [GSC15].

The AWARD support for dynamic reconfigurations is a distinctive contribution not yet supported in the widely used workflow systems. For instance, despite its constant development the Kepler system continues relying on a centralized execution engine and it does not offer any mechanism to perform dynamic reconfigurations of long-running workflows. However, we also have learned there is a need to better fulfill some still

open issues. In fact, this dissertation opened a set of directions for further improving the AWARD model and its implementation as well as the usability of the AWARD framework. Some of these research directions are pointed out in the following:

✧ **Unique names**

Currently in the AWARD framework the uniqueness of names of the activities and the input and output ports is ensured by the workflow developer. At execution time and in presence of dynamic reconfigurations to introduce new activities or input and output ports, uniqueness of names can be ensured by a global name service implemented using the AWARD Space.

✧ **Workflow hierarchies**

The AWARD model supports workflow hierarchies by encapsulating subworkflows within an activity *Task*. In fact, using the AWARD tools the *Task* performed by any workflow activity can launch another workflow. However, if a workflow has multiple iterations, the activity that encapsulates the subworkflow launches the entire subworkflow in each iteration which can be inefficient. Therefore there is a need of further developments to improve the efficiency of this functionality.

✧ **Dynamic reconfiguration operators**

The proposed set of dynamic reconfiguration operators was shown to be adequate for supporting useful and typical reconfiguration scenarios. However, other operators may be necessary according to specific requirements of the application cases.

✧ **Correctness issues**

The AWARD model only provides the basic mechanism for the activities to reach an agreement to choose the earliest global iteration between all activities involved such that the reconfiguration plan is applied. Therefore there is a need to address the verification of the correctness of AWARD workflows in dynamic reconfiguration scenarios.

✧ **Distributed AWARD Space**

The AWARD Space server implementation can replace the IBM TSpaces technology with a distributed tuple space implementation in order to avoid the occurrence of possible bottlenecks when the number of interactions between workflow activities increases.

Furthermore, a distributed implementation of the AWARD Space server can also increase significantly the storage capacity, boosting the unbounded size property of the AWARD Space to support a greater number of pending tokens, and tokens with a greater granularity.

✧ **Automate the generation of reconfiguration plans**

The process for submitting dynamic reconfiguration plans requires a Java program using the *DynamicLibrary.jar*. This can pose difficulties to some users to prepare and submit reconfigurations plans. Therefore the development of a new AWARD tool for interpreting a high-level script language or a graphical user interface for automatically generating Java reconfiguration plans can be useful to the end users.

✧ **Graphical user interface**

The AWARD framework tools need improvements in order to provide a flexible and user-friendly graphical interface able to design workflows, configure the necessary execution mappings, launch the workflow for execution, monitor and later submit dynamic workflow reconfigurations on parallel and distributed infrastructures.

✧ **Public availability of the AWARD framework**

A public virtual machine image of the AWARD framework is already available in the Amazon EC2 infrastructure that can be instantiated by others. However, in order to promote the dissemination of the AWARD model it is necessary to provide associated documentation.

7.4 Future Work

In the last decade there has been intensive research in scientific workflows. However, some issues and research challenges remain open. In addition to the above mentioned improvements to the AWARD model and its implementation, we also have identified the following challenges:

✧ **User interfaces for large-scale workflows**

The design of large-scale scientific workflow is itself a great challenge. However, the usability of user interfaces to promote the design of large-scale workflows characterized by a large number of activities is also an important challenge.

✧ **Big Data**

The integration of Big Data techniques and tools in data-flow workflows poses difficulties, for example, to avoid the movement of large amounts of data. In fact, Big Data issues are not just related to the storage size but also related to the composition of multiple distributed computing units working together in order to solve large problems.

✧ **Autonomic workflow behaviors**

Dynamic monitoring or mining the workflow provenance data can provide useful information to detect failures, or performance issues related to scalability. Therefore there is a need to support autonomic workflow behaviors to allow self-reconfigurations in order to recover from faults and support elastic scalability.

Although the importance of the above three challenges, we consider that the Big Data and autonomic behaviors challenges are more related to the improvement of the AWARD model. Therefore we select for future work the following two research questions:

1. How to integrate Big Data techniques and tools in the AWARD framework for allowing data-flow, where tokens and workflow activities manage large data sets;

2. How to improve the autonomic characteristics of the AWARD model for supporting self reconfigurations for instance automatic load balancing and fault recovery. This introduces the need for improving the AWARD workflow provenance data in order

to allow its dynamic monitoring and analysis. A possible direction can be the emergent concept and technologies related to micro services [New15].

Finally we would like to collaborate with users from other science domains in order to develop complex scientific applications that certainly could contribute to further validate and improve the AWARD model approach.

Bibliography

[AH00] W. Aalst and A. Hofstede. "Verification of Workflow Task Structures: A Petri-net-based Approach". In: *Information Systems* 25.1 (2000), pp. 43–69. DOI: 10.1016/S0306-4379(00)00008-9.

[AH05] W. Aalst and A. Hofstede. "YAWL: Yet Another Workflow Language". In: *Information Systems* 30.4 (2005), pp. 245 –275. DOI: 10.1016/j.is.2004.02.002.

[AHR11] W. Aalst, A. Hofstede, and N. Russell. *Workflow Patterns Web site*. 2011. URL: http://www.workflowpatterns.com/.

[APS09] W. Aalst, M. Pesic, and H. Schonenberg. "Declarative Workflows: Balancing between Flexibility and Support". In: *Computer Science - Research and Development* 23 (2 2009), pp. 99–113. DOI: 10.1007/s00450-009-0057-9.

[Aal+00a] W. Aalst, A. Barros, A. Hofstede, and B. Kiepuszewski. "Advanced Workflow Patterns". In: *Cooperative Information Systems*, *LNCS*. Vol. 1901. Springer Berlin / Heidelberg, 2000, pp. 18–29. DOI: 10.1007/10722620_2.

[Aal+00b] W. Aalst, A. Hofstede, B. Kiepuszewski, and A. Barros. "Workflow Patterns". In: BETA Working Paper Series WP 47, Eindhoven University of Technology. 2000, pp. 1–70.

[Aal+04] W. Aalst, L. Aldred, M. Dumas, and A. Hofstede. "Design and Implementation of the YAWL System". In: Procedings of 16th International Conference on Advanced Information Systems Engineering. 2004, pp. 142–159. DOI: 10.1007/978-3-540-25975-6_12.

[Aal+11] W. Aalst, K. Hee, A. Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. Wynn. "Soundness of Workflow Nets: Classification, Decidability, and Analysis". In: *Formal Aspects of Computing* 23.3 (May 2011), pp. 333–363. DOI: 10.1007/s00165-010-0161-4.

[AEA08] D. Abramson, C. Enticott, and I. Altintas. "Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, pp. 1–11. DOI: 10.1109/SC.2008.5215726.

[Add+03] M. Addis, J. Ferris, G. M., and et. "Experiences with e-Science Workflow Specification and Enactment in Bioinformatics". In: *Proceedings of e-Science All Hands Meeting*. 2003, pp. 459–466.

[Agh86] G. Agha. "Actors: A Model of Concurrent Computation in Distributed Systems". PhD thesis. Cambridge, MA, USA: MIT, 1986. ISBN: 0-262-01092-5.

[AB11] M. Agun and S. Bowers. "Approaches for Implementing Persistent Queues within Data-Intensive Scientific Workflows". In: *IEEE World Congress on Services*. 2011, pp. 200 –207. DOI: 10.1109/SERVICES.2011.57.

[AMA06] A. Akram, D. Meredith, and R. Allan. "Evaluation of BPEL to Scientific Workflows". In: *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06.* Vol. 1. May 2006, pp. 269 –274. DOI: 10.1109/CCGRID.2006.44.

[AC03a] M. Aksit and Z. Choukair. "Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision". In: *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*. 2003, pp. 84–89. DOI: 10.1109/ICDCSW.2003.1203537.

[AZE07] G. E. Allen, P. E. Zucknick, and B. L. Evans. "A Distributed Deadlock Detection and Resolution Algorithm for Process Networks". In: *IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*. Vol. 2. 2007, pp. 33–36. DOI: 10.1109/ICASSP.2007.366165.

[Alt+04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. "Kepler: An Extensible System for Design and Execution of Scientific Workflows". In: *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*. SSDBM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 423–424. DOI: 10.1109/SSDBM.2004.1311241.

[Ama+03] A. Amar, P. Boulet, J.-L. Dekeyser, and F. Theeuwen. *Distributed Process Networks Using Half FIFO Queues in CORBA*. Tech. rep. RR-4765. INRIA, 2003.

[Ama12] Amazon. *Amazon AWS - Amazon Elastic MapReduce (EMR)*. 2012. URL: http://aws.amazon.com/elasticmapreduce/.

[Ama13] Amazon. *Amazon AWS - EC2 Instance Images*. 2013. URL: https://aws.amazon.com/amis/.

[Ama15a] Amazon. *Amazon AWS - DynamoDB*. 2015. URL: http://aws.amazon.com/dynamodb/.

[Ama15b] Amazon. *Amazon AWS - Elastic Compute Cloud (EC2)*. 2015. URL: http://aws.amazon.com/ec2/.

[Apa15a] Apache. *Apache Hadoop*. 2015. URL: http://hadoop.apache.org/.

[Apa15b] Apache. *Apache Qpid, Messaging Built on AMQP*. 2015. URL: https://qpid.apache.org/.

[Arm+09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. U.C. Berkeley, 2009.

[AB84] Arvind and J. D. Brock. "Resource Managers in Functional Programming". In: *Journal of Parallel and Distributing Computing* 1.1 (1984), pp. 5–21.

[AC13] L. Assunção and J. C. Cunha. "Dynamic Workflow Reconfigurations for Recovering from Faulty Cloud Services". In: *IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom, 2013)*. 2013, pp. 88–95. DOI: 10.1109/CloudCom.2013.19.

[AC14] L. Assunção and J. C. Cunha. "Enabling Global Experiments with Interactive Reconfiguration and Steering by Multiple Users". In: *Proceedings of International Conference on Computational Science*. Ed. by Elsiever. Vol. 29. 2014, pp. 2137 –2144. DOI: 10.1016/j.procs.2014.05.198.

[AGC09] L. Assunção, C. Gonçalves, and J. C. Cunha. "On the Difficulties of Using Workflow Tools to Express Parallelism and Distribution - A Case Study in Geological Sciences". In: *Proceedings of the International Workshop on Workflow Management of the International Conference on Grid and Pervasive Computing (GPC2009)*. IEEE, 2009, pp. 104–110. DOI: 10.1109/GPC.2009.30.

[AGC12] L. Assunção, C. Gonçalves, and J. C. Cunha. "Autonomic Activities in the Execution of Scientific Workflows: Evaluation of the AWARD Framework". In: *Proceedings of the 9th IEEE International Conference on Autonomic and Trusted Computing (ATC 2012)*. 2012, pp. 423 –430. DOI: 10.1109/UIC-ATC.2012.14.

[AGC14] L. Assunção, C. Gonçalves, and J. C. Cunha. "Autonomic Workflow Activities: The AWARD Framework". In: *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)* 5.2 (2014), pp. 57–82. DOI: 10.4018/ijaras.2014040104.

[Atk08] A. Atkinson. "Tupleware: A Distributed Tuple Space for Cluster Computing". In: *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008.* 2008, pp. 121–126. DOI: 10.1109/PDCAT.2008.63.

[AC03b] V. Atluri and S. Chun. "Handling Dynamic Changes in Decentralized Workflow Execution Environments". In: *DEXA*. Ed. by V. Marík, W. Retschitzegger, and O. Stepánková. Vol. 2736. Lecture Notes in Computer Science. Springer, 2003, pp. 813–825. DOI: 10.1007/978-3-540-45227-0_79.

[BA04] R. Bajaj and D. P. Agrawal. "Improving Scheduling of Tasks in a Heterogeneous Environment". In: *IEEE Transactions on Parallel and Distributed Systems* 15.2 (2004), pp. 107–118. DOI: 10.1109/TPDS.2004.1264795.

[Bal+14] J. Balderrama, M. Simonin, L. Ramakrishnan, V. Hendrix, C. Morin, D. Agarwal, and C. Tedeschi. "Combining Workflow Templates with a Shared Space-based Execution Model". In: *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*. WORKS '14. New Orleans, Louisiana: IEEE Press, 2014, pp. 50–58. DOI: 10.1109/WORKS.2014.14.

[BG07] R. Barga and D. Gannon. "Workflows for e-Science: Scientific Workflows for Grids". In: ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer, 2007. Chap. Scientific versus Business Workflows, pp. 9–16. DOI: 10.1007/978-1-84628-757-2_2.

[BB11] A. Barker and R. Buyya. "Decentralised Orchestration of Service-Oriented Scientific Workflows". In: *1st International Conference on Cloud Computing and Services Science*. 2011, pp. 222–231.

[BH08] A. Barker and J. Hemert. "Scientific Workflow: A Survey and Research Directions". In: *LNCS - Parallel Processing and Applied Mathematics*. Vol. 4967. Springer Berlin, 2008, pp. 746–753. DOI: 10.1007/978-3-540-68111-3_78.

[BWH09] A. Barker, J. B. Weissman, and J. I. Hemert. "The Circulate Architecture: Avoiding Workflow Bottlenecks Caused by Centralised Orchestration". In: *Cluster Computing* 12.2 (2009), pp. 221–235. DOI: 10.1007/s10586-009-0072-4.

[BH01] T. Basten and J. Hoogerbrugge. "Efficient Execution of Process Networks". In: *Proceedings of Communicating Process Architectures*. IOS Press, 2001, pp. 1–14.

[Ben+12] J. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. "Combining In-situ and In-transit Processing to Enable Extreme-Scale Scientific Analysis". In: *International Conference for High Performance Computing, Network, Storage and Analysis*. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[Ben+13] A. Benoit, U. V. Çatalyürek, Y. Robert, and E. Saule. "A Survey of Pipelined Workflow Scheduling: Models and Algorithms". In: *ACM Computing Surveys* 45.4 (2013), 50:1–50:36. DOI: 10.1145/2501654.2501664.

[BC92] L. Bernardinello and F. d. Cindio. "A Survey of Basic Net Models and Modular Net Classes". In: *Advances in Petri Nets 1992, The DEMON Project*. London, UK, UK: Springer-Verlag, 1992, pp. 304–351. DOI: 10.1007/3-540-55610-9_177.

[BL05] S. Bowers and B. Ludascher. "Actor-Oriented Design of Scientific Workflows". In: *LNCS- In 24th International Conference on Conceptual Modeling*. Ed. by Springer. Vol. 3716. Springer Berlin / Heidelberg, 2005, pp. 369–384. DOI: 10.1007/11568322_24.

[BPE06] T. C. BPEL. *Business Process Execution Language for Web Services*. Tech. rep. OASIS, 2006. URL: https://www.oasis-open.org/.

[Bra+04] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. "A Survey of Self-management in Dynamic Software Architecture Specifications". In: *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*. WOSS '04. ACM, 2004, pp. 28–33. DOI: 10.1145/1075405.1075411.

[BL08] C. Brooks and E. Lee. *Heterogeneous Concurrent Modeling and Design in Java (Ptolemy II Software, Technical Reports: UCB/EECS-2008-28, UCB/EECS-2008-29, UCB/EECS-2008-37)*. Tech. rep. Berkeley: University of California, 2008, Vol. 1–3.

[Bro+08a] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*. Tech. rep. University of California at Berkeley, 2008.

[Bro+08b] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*. Tech. rep. University of California at Berkeley, 2008.

[Bru07] P. Brucker. *Scheduling Algorithms*. 5th Edition. Springer, 2007. ISBN: 978-3-540-69515-8.

[BL10] T. Burkhart and P. Loos. "Flexible Business Processes - Evaluation of Current Approaches". In: *Proceedings of Multikonferenz Wirtschaftsinformatik (MKWI-2010)*. 2010, pp. 1217–1228.

[BL13] M. Bux and U. Leser. "Parallelization in Scientific Workflow Management Systems". In: *The American Computing Research Repository* abs/1303.7195 (2013).

[CNP08] M. Caeiro, Z. Nemeth, and T. Priol. "A Chemical Workflow Engine for Scientific Workflows with Dynamicity support". In: *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*. 2008, pp. 1–10. DOI: 10.1109/WORKS.2008.4723956.

[CG89] N. Carriero and D. Gelernter. "Linda in Context". In: *Communication of the ACM* 32.4 (1989), pp. 444–458. DOI: 10.1145/63334.63337.

[CD12] W. Chen and E. Deelman. "Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints". In: *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part II*. PPAM'11. Torun, Poland: Springer-Verlag, 2012, pp. 11–20. DOI: 10.1007/978-3-642-31500-8_2.

[Che+15] W. Chen, R. F. da Silva, E. Deelman, and R. Sakellariou. "Using Imbalance Metrics to Optimize Task Clustering in Scientific Workflow Executions". In: *Future Generation Computer Systems* 46 (2015), pp. 69 –84. DOI: 10.1016/j.future.2014.09.014.

[Chi+11] G. Chin, C. Sivaramakrishnan, T. Critchlow, K. Schuchardt, and A. Ngu. "Scientist-Centered Workflow Abstractions via Generic Actors, Workflow Templates, and Context-Awareness for Groundwater Modeling and Analysis". In: *IEEE Congress on Services* (2011), pp. 176–183. DOI: 10.1109/SERVICES.2011.31.

[Chu+06] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, and M. Shields. "Programming Scientific and Distributed Workflow with Triana Services". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1021–1037. DOI: 10.1002/cpe.v18:10.

[Col12] Collide. *COLLIDE - Faculty of Engineering, University of Duisburg-Essen, Germany*. 2012. URL: http://sqlspaces.collide.info/.

[Cot+07] R. G. Cote, P. Jones, L. Martens, S. Kerrien, F. Reisinger, Q. Lin, R. Leinonen, R. Apweiler, and H. Hermjakob. "The Protein Identifier Cross-Referencing Service: Reconciling Protein Identifiers across Multiple Source Databases". In: *BMC Bioinformatics* 8 (1 2007), pp. 1–14. DOI: 10.1186/1471-2105-8-401.

[Cou+07] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. "Workflows for e-Science: Scientific Workflows for Grids". In: ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. Workflow Management in Condor, pp. 357–375.

[CG08] V. Curcin and M. Ghanem. "Scientific Workflow Systems - Can One Size Fit All?" In: *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*. 2008, pp. 1–9. DOI: 10.1109/CIBEC.2008.4786077.

[DR09] P. Dadam and M. Reichert. "The ADEPT Project: a Decade of Research and Development for Robust and Flexible Process Support". In: *Computer Science* 23 (2 2009), pp. 81–97.

[DK07] M. I. Daoud and N. Kharma. "A high Performance algorithm for Static Task Scheduling in Heterogeneous Distributed Computing Systems". In: *Journal of Parallel and Distributed Computing* 68.4 (2007), pp. 399 –409. DOI: 10.1016/j.jpdc.2007.05.015.

[DF08] S. B. Davidson and J. Freire. "Provenance and Scientific Workflows: Challenges and Opportunities". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1345–1350. DOI: 10.1145/1376616.1376772.

[DG04] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[Dee07] E. Deelman. "Looking into the Future of Workflows: The Challenges Ahead". In: *Workflows for e-Science: Scientific Workflows for Grids*. Ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. 28, pp. 475–481. DOI: 10.1007/978-1-84628-757-2_28.

[Dee+05] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. "Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems". In: *Scientific Programming* 13 (3 2005), pp. 219–237.

[Dee+08] E. Deelman, D. Gannon, M. Shields, and I. Taylor. "Workflows and e-Science: An Overview of Workflow System Features and Capabilities". In: *Future Generation Computer Systems* 25.5 (2008), pp. 528–540.

[Dee+15] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. "Pegasus, a Workflow Management System for Science Automation". In: *Future Generation Computer Systems* 46.C (May 2015), pp. 17–35. DOI: 10.1016/j.future.2014.10.008.

[Dee+16] E. Deelman, K. Vahi, M. Rynge, G. Juve, R. Mayani, and R. F. da Silva. "Pegasus in the Cloud: Science Automation through Workflow Technologies". In: *Internet Computing, IEEE* 20.1 (2016), pp. 70–76. ISSN: 1089-7801. DOI: 10.1109/MIC.2016.15.

[Doo95] R. B. Doorenbos. "Production Matching for Large Learning Systems". UMI Order No. GAX95-22942. PhD thesis. Pittsburgh, PA, USA, 1995.

[Dru09] C. Drummond. "Replicability is not Reproducibility: Nor is it good science". In: *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*. 2009.

[EBI12] EBI. *Protein Identifier Cross-Reference Service*. European Bioinformatics Institute. Oct. 2012. URL: http://www.ebi.ac.uk/Tools/picr/.

[EHT10] E. Elmroth, F. Hernández, and J. Tordsson. "Three Fundamental dimensions of scientific Workflow Interoperability: Model of Computation, Language, and Execution Environment". In: *Future Generation Computer Systems* 26 (2 2010), pp. 245–256.

[Fad+12] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju. "MARLA: MapReduce for Heterogeneous Clusters". In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 49–56. DOI: 10.1109/CCGrid.2012.135.

[FQH05] T. Fahringer, J. Qin, and S. Hainzer. "Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language". In: vol. 2. IEEE International Symposium on Cluster Computing and the Grid. 2005, pp. 676–685.

[FTP11] H. Fernandez, C. Tedeschi, and T. Priol. "A Chemistry-Inspired Workflow Management System for Scientific Applications in Clouds". In: *IEEE 7th International Conference on E-Science*. 2011, pp. 39 –46. DOI: 10.1109/eScience.2011.14.

[For90] C. L. Forgy. "Expert Systems". In: ed. by P. G. Raeth. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. Chap. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pp. 324–341.

[FK03] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 1558609334.

[FH10] E. Friedman-Hill. *Jess: the Rule Engine for the Java Platform*. 2010. URL: http://www.jessrules.com/jess/.

[GHR94] E. Gallopoulos, E. Houstis, and J. R. Rice. "Computer as Thinker/Doer: Problem-solving Environments for Computational Science". In: *IEEE Computational Science and Engineering* 1.2 (1994), pp. 11–23. DOI: 10.1109/99.326669.

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Gar+12] D. Garijo, P. Alper, K. Belhajjame, O. Corcho, Y. Gil, and C. Goble. "Common Motifs in Scientific Workflows: An Empirical Analysis". In: *IEEE 8th International Conference on E-Science (e-Science, 2012)*. 2012, pp. 1–8. DOI: 10.1109/eScience.2012.6404427.

[GGC14] D. Garijo, Y. Gil, and O. Corcho. "Towards Workflow Ecosystems Through Semantic and Standard Representations". In: *Proceedings of the Ninth Workshop on Workflows in Support of Large-Scale Science (WORKS), held in conjunction with the IEEE ACM International Conference on High-Performance Computing (SC)*. New Orleans, LA, 2014.

[GB03] M. Geilen and T. Basten. "Requirements on the Execution of Kahn Process Networks". In: *Proceedings of the 12th European Conference on Programming*. Vol. 2618. Warsaw, Poland: Springer-Verlag, 2003, pp. 319–334. DOI: 10.1007/3-540-36575-3_22.

[Gig12] Gigaspaces. *XAP 8.0 Documentation, Gigaspaces.com*. Gigaspaces. 2012. URL: http://www.gigaspaces.com/wiki/display/XAP8/8.0.

[Gil+07] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. "Examining the Challenges of Scientific Workflows". In: *IEEE Computer* 40.12 (2007), pp. 24–32. DOI: 10.1109/MC.2007.421.

[Gil+11] Y. Gil, V. Ratnakar, J. Kim, J. Moody, E. Deelman, P. Gonzalez-Calero, and P. Groth. "Wings: Intelligent Workflow-Based Design of Computational Experiments". In: *Intelligent Systems, IEEE* 26.1 (2011), pp. 62–72. DOI: 10.1109/MIS.2010.9.

[GR16] Y. Gil and V. Ratnakar. "Dynamically Generated Metadata and Replanning by Interleaving Workflow Generation and Execution". In: *Proceedings of the Tenth IEEE International Conference on Semantic Computing (ICSC)*. Irvine, CA, 2016.

[GWS03] C. Goble, C. Wroe, and R. Stevens. "The myGrid Project: Services, Architecture and Demonstrator". In: *UK e-Science All Hands Meeting*. 2003.

[God+07] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. "Composing Different Models of Computation in Kepler and Ptolemy II". In: *Proceedings of the 7th International Conference on Computational Science, Part III: ICCS 2007*. Beijing, China: Springer-Verlag, 2007, pp. 182–190. DOI: 10.1007/978-3-540-72588-6_33.

[God+09] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. "Heterogeneous Composition of Models of Computation". In: *Future Generation Computer Systems* 25 (5 2009), pp. 552–560.

[Goe+10] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. "Galaxy: A Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences". In: *Genome Biology* 11 (2010).

[GRC08] C. Gomes, O. F. Rana, and J. Cunha. "Extending Grid-Based Workflow Tools With Patterns/Operators". In: *International Journal of High Performance Computing Applications* 22 (3 2008), pp. 301–318.

[Gon17] C. Gonçalves. "A Problem Solving Environment for Parallel Extraction of Multiwords and Applications from Large Corpora". PhD thesis. To be presented at FCT-UNL, 2017.

[GAC12] C. Gonçalves, L. Assunção, and J. C. Cunha. "Data Analytics in the Cloud with Flexible MapReduce Workflows". In: *IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom, 2012)*. 2012, pp. 427 –434. DOI: 10.1109/CloudCom.2012.6427527.

[GAC13] C. Gonçalves, L. Assunção, and J. C. Cunha. "Flexible MapReduce Workflows for Cloud Data Analytics". In: *International Journal of Grid and High Performance Computing (IJGHPC)* 5.4 (2013). IGI Global, pp. 48–64. DOI: doi:10.4018/ijghpc.2013100104.

[GSC15] C. Gonçalves, J. F. Silva, and J. C. Cunha. "A Parallel Algorithm for Statistical Multiword Term Extraction from Very Large Corpora". In: *International Conference on High Performance Computing and Communications (HPCC), 2015*. IEEE, 2015, pp. 219–224. DOI: 10.1109/HPCC-CSS-ICESS.2015.72.

[Gon+13] J. Goncalves, D. Oliveira, K. Ocana, E. Ogasawara, J. Dias, and M. Mattoso. "Performance Analysis of Data Filtering in Scientific Workflows". In: *Journal of Information and Data Management* 4.1 (2013), pp. 17–26.

[Goo15] Google. *Google Translator API*. 2015. URL: https://developers.google.com/translate/.

[Gub+06] T. Gubala, D. Herezlak, M. Bubak, and M. Malawski. "Semantic Composition of Scientific Workflows Based on the Petri Nets Formalism". In: *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. IEEE Computer Society, 2006, p. 12. DOI: 10.1109/E-SCIENCE.2006.127.

[Gun+10] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. "MapReduce in the Clouds for Science". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 2010, pp. 565 –572. DOI: 10.1109/CloudCom.2010.107.

[HA00] C. Hagen and G. Alonso. "Exception Handling in Workflow Management Systems". In: *IEEE Transactions on Software Engineering* 26.10 (2000), pp. 943–958. DOI: 10.1109/32.879818.

[HSW01] J. Halliday, S. Shrivastava, and S. Wheater. "Flexible Workflow Management in the OPENflow System". In: *Proceedings of Fifth IEEE International Enterprise Distributed Object Computing Conference EDOC '01*. 2001, pp. 82 –92. DOI: 10.1109/EDOC.2001.950425.

[HPA05] T. Heinis, C. Pautasso, and G. Alonso. "Design and Evaluation of an Autonomic Workflow Engine". In: *Proceedings of Second International Conference on Autonomic Computing*. 2005, pp. 27–38.

[Hei+99] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. "A Comprehensive Approach to Flexibility in Workflow Management Systems". In: *ACM SIGSOFT Software Engineering Notes* 24 (2 1999), pp. 79–88. DOI: 10.1145/295666.295675.

[HT96] R. Henderson and D. Tweten. *Portable Batch System: External Reference Specification*. Tech. rep. NASA, Ames Research Center, 1996.

[HKR13] N. R. Herbst, S. Kounev, and R. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not". In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[Hin06] C. M. Hine. *New Infrastructures for Knowledge Production: Understanding E-science*. Arlington, VA, USA: Information Resources Press, 2006. ISBN: 1591407184.

[Hoh06] A. Hoheisel. "User Tools and Languages for Graph-based Grid Workflows". In: *Concurrency and Computation: Practice and Experience* (2006), pp. 1101–1113. DOI: 10.1002/cpe.1002.

[HA07] A. Hoheisel and M. Alt. "Petri Nets". In: *Workflows for e-Science: Scientific Workflows for Grids*. Ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. 13, pp. 190–207.

[Hol95] D. Hollingsworth. *Workflow Management Coalition - The Workflow Reference Model*. Tech. rep. Workflow Management Coalition, 1995.

[How12] B. Howe. "Virtual Appliances, Cloud Computing, and Reproducible Research". In: *Computing in Science Engineering* 14.4 (2012), pp. 36–41. DOI: 10.1109/MCSE.2012.62.

[HK03] S. Hwang and C. Kesselman. "Grid workflow: A Flexible Failure Handling Framework for the Grid". In: *Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing*. 2003, pp. 126–137. DOI: 10.1109/HPDC.2003.1210023.

[IEE12] IEEE. "IEEE Systems and Software Engineering - Architecture Description". In: *Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000* (2012), pp. 1–46. DOI: 10.1109/IEEESTD.2011.6129467.

[IT07] E. Ilavarasan and P. Thambidurai. "Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments". In: *Journal of Computer Sciences* 3(2) (2007), pp. 94–103.

[Jav15] Java Community Process. *Java Message Service Specification*. 2015. URL: http://www.oracle.com/technetwork/java/docs-136352.html.

[JPW00] C. R. Johnson, S. G. Parker, and D. Weinstein. "Large-Scale Computational Science Applications Using the SCIRun Problem Solving Environment". In: *Supercomputer*. 2000, p. 19.

[Juv+10] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. "Data Sharing Options for Scientific Workflows on Amazon EC2". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–9.

[Kah74] G. Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information processing*. Ed. by J. L. Rosenfeld. 1974, pp. 471–475.

[Kaj+09] T. Kajiyama, D. Alimonte, J. C. Cunha, and G. Zibordi. "High-Performance Ocean Color Monte Carlo Simulation in the Geo-info Project". In: *8th International Conference on Parallel Processing and Applied Mathematics, (PPAM 2009)*. 2009, pp. 370–379. DOI: 10.1007/978-3-642-14403-5_39.

[Kam+00] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman. "Techniques for Supporting Dynamic and Adaptive Workflow". In: *Computer Supported Cooperative Work* 9 (3-4 2000), pp. 269–292. DOI: 10.1023/A:1008747109146.

[Kep05] J. Kephart. "Research Challenges of Autonomic Computing". In: *Proceedings of 27th International Conference on Software Engineering*. 2005, pp. 15–22. DOI: 10.1109/ICSE.2005.1553533.

[KC03] J. Kephart and D. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (2003), pp. 41 –50. DOI: 10.1109/MC.2003.1160055.

[Kep13] Kepler project. *Kepler web site*. 2013. URL: https://kepler-project.org/.

[Kep14] Kepler project. *Kepler User Manual*. 2014, pp. 97–. URL: https://kepler-project.org/.

[KHA03] B. Kiepuszewski, A. Hofstede, and W. Aalst. "Fundamentals of Control Flow in Workflows". In: *Acta Informatica* 39.3 (2003), pp. 143–209. DOI: 10.1007/s00236-002-0105-4.

[KM85] J. Kramer and J. Magee. "Dynamic Configuration for Distributed Systems". In: *IEEE Transactions on Software Engineering* 11.4 (Apr. 1985), pp. 424–436. DOI: 10.1109/TSE.1985.232231.

[KM09] J. Kramer and J. Magee. "A Rigorous Architectural Approach to Adaptive Software Engineering". In: *Journal of Computer Science and Technology* 24.2 (2009), pp. 183–188. DOI: 10.1007/s11390-009-9216-5.

[KA99] Y.-K. Kwok and I. Ahmad. "Benchmarking and Comparison of the Task Graph Scheduling Algorithms". In: *Journal of Parallel and Distributed Computing* 59.3 (Dec. 1999), pp. 381–422. DOI: 10.1006/jpdc.1999.1578.

[Lac+10] M. Lackovic, D. Talia, R. Tolosana-Calasanz, J. Banares, and O. Rana. "A Taxonomy for the Analysis of Scientific Workflow Faults". In: *IEEE 13th International Conference on Computational Science and Engineering (CSE, 2010)*. 2010, pp. 398 –403. DOI: 10.1109/CSE.2010.59.

[LM10] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[Laz11] G. Lazweski. *Scientific Workflow Survey*. 2011. URL: http://wiki.cogkit.org/index.php/Scientific_Workflow_Survey.

[LP95] E. Lee and T. Parks. "Dataflow Process Networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773 –801. DOI: 10.1109/5.381846.

[Leh+01] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. "Hitting the Distributed Computing Sweet Spot with TSpaces". In: *Computer Networks* 35.4 (2001), pp. 457–472.

[LMJ10] G. Li, V. Muthusamy, and H. A. Jacobsen. "A Distributed Service-oriented Architecture for Business Process Execution". In: *ACM Trans. Web* 4.1 (Jan. 2010), 2:1–2:33. DOI: 10.1145/1658373.1658375.

[Li+06] G. Li, V. Muthusamy, H.-A. Jacobsen, and S. Mankovski. "Decentralized Execution of Event-Driven Scientific Workflows". In: *IEEE Services Computing Workshops* (2006), pp. 73–82. DOI: 10.1109/SCW.2006.10.

[LP05] Z. Li and M. Parashar. "Comet: A Scalable Coordination Space for Decentralized Distributed Environments". In: *Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P)*. 2005, pp. 104–112.

[LP06] Z. Li and M. Parashar. "An Infrastructure for Dynamic Composition of Grid Services". In: *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*. GRID '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 315–316. DOI: 10.1109/ICGRID.2006.311036.

[LP07] Z. Li and M. Parashar. *An Infrastructure for Dynamic Composition of Grid Services*. Tech. rep. Rutgers University, 2007.

[LGX10] D. Liu, T. Gu, and J.-P. Xue. "Rule Engine Based on Improvement Rete Algorithm". In: *International Conference on Apperceiving Computing and Intelligence Analysis (ICACIA, 2010)*. 2010, pp. 346–349. DOI: 10.1109/ICACIA.2010.5709916.

[Liu+07] X. Liu, W. Dou, J. Chen, S. Fan, S. Cheung, and S. Cai. "On Design, Verification, and Dynamic Modification of the Problem-based Scientific Workflow Model". In: *Simulation Modelling Practice and Theory* 15.9 (2007), pp. 1068–1088. DOI: 10.1016/j.simpat.2007.06.003.

[Lud+06] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, E. Jones, A. Lee, J. Tao, and Y. Zhao. "Scientific Workflow Management and the KEPLER System". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[Lun15] LunaCloud. 2015. URL: http://www.lunacloud.com/.

[LGD15] C. M. Lushbough, E. Z. Gnimpieba, and R. Dooley. "Life Science Data Analysis Workflow Development Using the Bioextract Server Leveraging the iPlant Collaborative Cyberinfrastructure". In: *Concurrency and Computation: Practice and Experience* 27.2 (2015), pp. 408–419. DOI: 10.1002/cpe.3237.

[MTB13] Y. Mansouri, A. N. Toosi, and R. Buyya. "Brokering Algorithms for Optimizing the Availability and Cost of Cloud Storage Services". In: *IEEE 5th International Conference on Cloud Computing Technology and Science*. 2013, pp. 581–589. DOI: 10.1109/CloudCom.2013.19.

[MWL08a] D. Martin, D. Wutke, and F. Leymann. "A Novel Approach to Decentralized Workflow Enactment". In: *IEEE 12th International Conference Enterprise Distributed Object Computing (EDOC'08)*. 2008, pp. 127–136. DOI: 10.1109/EDOC.2008.22.

[MWL08b] D. Martin, D. Wutke, and F. Leymann. "Using Tuplespaces to Enact Petri Net-based Workflow Definitions". In: *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. Linz, Austria: ACM, 2008, pp. 63–70. DOI: 10.1145/1497308.1497324.

[Mat+13] M. Mattoso, K. Ocana, F. Horta, J. Dias, E. Ogasawara, V. Silva, D. Oliveira, F. Costa, and A. Igor. "User-steering of HPC Workflows: State-of-the-art and Future Directions". In: *2nd ACM Workshop on Scalable Workflow Execution Engines and Technologies*. New York, 2013, 4:1–4:6. DOI: 10.1145/2499896.2499900.

[Mat+15] M. Mattoso, J. Dias, K. A. Ocaña, E. Ogasawara, F. Costa, F. Horta, V. Silva, and D. de Oliveira. "Dynamic Steering of HPC Scientific Workflows: A Survey". In: *Future Generation Computer Systems* 46 (2015), pp. 100 –113. DOI: 10.1016/j.future.2014.11.017.

[McP+09] T. McPhillips, S. Bowers, D. Zinn, and B. Ludascher. "Scientific Workflow Design for Mere Mortals". In: *Future Generation Computer Systems* 25.5 (2009), pp. 541–551. DOI: 10.1016/j.future.2008.06.013.

[Mic15] Microsoft. *Bing Microsoft Translator Web Service*. 2015. URL: http://www.microsoft.com/web/post/using-the-free-bing-translation-apis.

[Mig+11] S. Migliorini, M. Gambini, M. Rosa, and A. Hofstede. *Pattern-based Evaluation of Scientific Workflow Management Systems*. Tech. rep. University of Verona, Italy and Queensland University of Technology, Australia, 2011.

[Mil+08] S. Miles, P. Groth, E. Deelman, K. Vahi, G. Mehta, and L. Moreau. "Provenance: The Bridge Between Experiments and Data". In: *Computing in Science and Engineering* 10.3 (2008), pp. 38–46.

[Mis+10] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. "Taverna, Reloaded". In: *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management*. Heidelberg, Germany: Springer-Verlag, 2010, pp. 471–481.

[MFPF12] H. Mohammadi Fard, R. Prodan, and T. Fahringer. "A Truthful Dynamic Workflow Scheduling Mechanism for Commercial Multi-Cloud Environments". In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2012), pp. 1203 –1212. DOI: 10.1109/TPDS.2012.257.

[Mon15] Montage. *Montage toolkit*. CALTECH/JPL. 2015. URL: http://montage.ipac.caltech.edu/.

[Mur89] T. Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. ISSN: 0018-9219. DOI: 10.1109/5.24143.

[NCS04] M. G. Nanda, S. Chandra, and V. Sarkar. "Decentralizing Execution of Composite Web Services". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2004, pp. 170–187. DOI: 10.1145/1028976.1028991.

[NeS12] NeSC. *UK e-Science Centres*. 2012. URL: http://www.nesc.ac.uk/centres/.

[New15] S. Newman. *Building Microservices - Designing Fine-Grained Systems*. Ed. by M. Loukides and B. MacDonald. O´Reilly, 2015.

[Ngu+08] A. Ngu, S. Bowers, N. Haasch, T. Mcphillips, and T. Critchlow. "Flexible Scientific Workflow Modeling Using Frames, Templates, and Dynamic Embedding". In: *Proceedings of the 20th international conference on Scientific and Statistical Database Management*. Springer-Verlag, 2008, pp. 566–572.

[Ngu+15] H. Nguyen, D. Abramson, T. Kipouros, A. Janke, and G. Galloway. "WorkWays: Interacting with Scientific Workflows". In: *Concurrency and Computation: Practice and Experience* 27.16 (2015), pp. 4377–4397. DOI: 10.1002/cpe.3525.

[NPP05] Z. Németh, C. Pérez, and T. Priol. "Workflow Enactment Based on a Chemical Metaphor". In: 3rd IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society Press, 2005, pp. 127–136. DOI: 10.1109/SEFM.2005.55.

[NPP06] Z. Németh, C. Pérez, and T. Priol. "Distributed Workflow Coordination: Molecules and Reactions". In: 20th International Parallel and Distributed Processing Symposium. 2006. DOI: 10.1109/IPDPS.2006.1639517.

[OAS15a] OASIS. *AdvancedMessageQueuingProtocol (AMQP)*. 2015. URL: https://www.amqp.org/.

[OAS15b] OASIS. *Business Process Execution Language (BPEL)*. OASIS. 2015. URL: http://bpel.xml.org/about-bpel.

[OG02] P. Obreiter and G. Graf. "Towards Scalability in Tuple Spaces". In: *Proceedings of the 2002 ACM symposium on Applied computing (SAC´02)*. New York, NY, USA: ACM, 2002, pp. 344–350. DOI: 10.1145/508791.508858.

[Oli+06] T. Olinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, and C. Goble. "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1067–1100.

[OE06] A. G. Olson and B. L. Evans. "Deadlock Detection for Distributed Process Networks". In: *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*. 2006, pp. 73–76. DOI: 10.1.1.122.5340.

[OMG15] OMG. *Unified Modeling Language*. 2015. URL: http://www.uml.org/.

[Ora12] Oracle. *Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms*. 2012. URL: http://www.oracle.com/technetwork/articles/javase/javaspaces-140665.html.

[Oro+11] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao. "TIDeFlow: The Time Iterated Dependency Flow Execution Model". In: *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM, 2011)*. 2011, pp. 1–9. DOI: 10.1109/DFM.2011.11.

[PJ95] S. Parker and C. Johnson. "SCIRun: A Scientific Programming Environment for Computational Steering". In: *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. 1995, p. 52. DOI: 10.1109/SUPERC.1995.241689.

[Par95] T. M. Parks. "Bounded Scheduling of Process Networks". PhD thesis. University of California at Berkeley, 1995.

[PR03] T. M. Parks and D. Roberts. "Distributed Process Networks in Java". In: *Proceedings of International Parallel and Distributed Processing Symposium*. 2003, pp. 8–16. DOI: 10.1109/IPDPS.2003.1213266.

[PA04] C. Pautasso and G. Alonso. "JOpera: a Toolkit for Efficient Visual Composition of Web Services". In: *International Journal of Electronic Commerce (IJEC)* 9 (2004), pp. 107–141.

[Plo+13] M. Plociennik, T. Zok, I. Altintas, J. Wang, D. Crawl, D. Abramson, F. Imbeaux, B. Guillerminet, M. Lopez-Caniego, I. C. Plasencia, W. Pych, P. Ciecieląg, B. Palak, M. Owsiak, and Y. Frauel. "Approaches to Distributed Execution of Scientific Workflows in Kepler". In: *Fundamenta Informaticae* 128.3 (2013), pp. 281–302. DOI: 10.3233/FI-2013-947.

[RG08] L. Ramakrishnan and D. Gannon. *A Survey of Distributed Workflow Characteristics and Resource Requirements*. Tech. rep. TR671. Indiana University, 2008.

[RP10] L. Ramakrishnan and B. Plale. "A Multi-dimensional Classification Model for Scientific Workflow Characteristics". In: *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*. Indianapolis, Indiana: ACM, 2010, 4:1–4:12. DOI: 10.1145/1833398.1833402.

[RB07] M. Reichert and T. Bauer. "Supporting Ad-hoc Changes in Distributed Workflow Management Systems". In: *Proceedings of the 2007 OTM Confederated International Conference on the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS*. Springer-Verlag, 2007, pp. 150–168.

[RD97] M. Reichert and P. Dadam. "A Framework for Dynamic Changes in Workflow Management Systems". In: *Proceedings of DEXA'97*. IEEE Computer Society, 1997, pp. 42 –48. DOI: 10.1109/DEXA.1997.617231.

[RD98] M. Reichert and P. Dadam. "Adeptflex - Supporting Dynamic Changes of Workflows Without Losing Control". In: *Journal of Intelligent Information Systems* 10.2 (1998), pp. 93–129. DOI: 10.1023/A:1008604709862.

[RRD04] S. Rinderle, M. Reichert, and P. Dadam. "Correctness Criteria for Dynamic Changes in Workflow Systems: A Survey". In: *Data and Knowledge Engineering* 50.1 (July 2004), pp. 9–34. DOI: 10.1016/j.datak.2004.01.002.

[RAH06] N. Russell, W. Aalst, and A. Hofstede. "Workflow Exception Patterns". In: *Advanced Information Systems Engineering*. Ed. by E. Dubois and K. Pohl. Vol. 4001. Springer, 2006, pp. 288–302. DOI: 10.1007/11767138_20.

[RH09] N. Russell and A. Hofstede. "Surmounting BPM Challenges: The YAWL Story". In: *Computer Science - Research and Development* 23 (2 2009), pp. 67–79. DOI: 10.1007/s00450-009-0059-7.

[RHE05] N. Russell, A. Hofstede, and D. Edmond. "Workflow Resource Patterns: Identification, Representation and Tool Support". In: vol. 3520. Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAISE). LNCS, Springer, 2005, pp. 216–232. DOI: 10.1007/11431855_16.

[Rus+05] N. Russell, A. Hofstede, D. Edmond, and W. Aalst. "Workflow Data Patterns: Identification, Representation and Tool Support". In: *Proceedings of the 24th International Conference on Conceptual Modeling*. Springer-Verlag, 2005, pp. 353–368. ISBN: 3-540-29389-2, 978-3-540-29389-7. DOI: 10.1007/11568322_23.

[San+88] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. "Innovations in Internetworking". In: ed. by C. Partridge. Norwood, MA, USA: Artech House, Inc., 1988. Chap. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[SP+16] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho. "Reproducibility of Execution Environments in Computational Science Using Semantics and Clouds". In: *Future Generation Computer Systems* (2016). DOI: 10.1016/j.future.2015.12.017.

[Shi07a] M. Shields. "Workflows for e-Science: Scientific Workflows for Grids". In: ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. The Triana Workflow Environment: Architecture and Applications, pp. 320–339.

[Shi07b] M. Shields. "Workflows for e-Science: Scientific Workflows for Grids". In: ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. Control- Versus Data-Driven Workflows, pp. 167–173.

[SC15] J. F. Silva and J. C. Cunha. *A Problem Solving Environment for Parallel Extraction of Multiwords and Applications from Large Corpora*. 2015. URL: http://hlt.di.fct.unl.pt/jfs/index.htm\#Research.

[Sil+99] J. F. Silva, G. Dias, S. Guillore, and J. G. P. Lopes. "Using LocalMaxs Algorithm for the Extraction of Contiguous and Non-contiguous Multiword Lexical Units". In: *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*. EPIA '99. Springer-Verlag, 1999, pp. 113–132.

[Slo07] A. Slominski. "Adapting BPEL to Scientific Workflows ". In: ed. by I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. Springer-Verlag London, 2007. Chap. 14, pp. 208–226.

[SLI08] S. Smanchat, S. Ling, and M. Indrawan. "A Survey on Context-aware Workflow Adaptations". In: *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*. MoMM '08. New York, NY, USA: ACM, 2008, pp. 414–417.

[SKD10] M. Sonntag, D. Karastoyanova, and E. Deelman. "Bridging the Gap between Business and Scientific Workflows: Humans in the Loop of Scientific Workflows". In: *IEEE Sixth International Conference on e-Science (e-Science), 2010*. 2010, pp. 206–213. DOI: 10.1109/eScience.2010.12.

[Sor+07] S. W. Sorde, S. K. Aggarwal, J. Song, M. Koh, and S. See. "Modeling and Verifying Non-DAG Workflows for Computational Grids". In: *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 237–243.

[Sou+04] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. S. Rosenburg, and J. Xenidis. "System Support for Online Reconfiguration". In: *USENIX Annual Technical Conference, General Track*. USENIX, Jan. 5, 2004, pp. 141–154.

[Spa12] Space Based Computing Group. *MozartSpaces - Java implementation of eXtensible Virtual Shared Memory (XVSM)*. Vienna University of Technology, Institute of Computer Languages. 2012. URL: http://www.mozartspaces.org/2.3-SNAPSHOT/.

[SS13] N. Stojnic and H. Schuldt. "OSIRIS-SR: A Scalable Yet Reliable Distributed Workflow Execution Engine". In: *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. SWEET '13. ACM, 2013, 3:1–3:12. DOI: 10.1145/2499896.2499899.

[SJ09] P. Sun and C. Jiang. "Analysis of Workflow Dynamic Changes Based on Petri Net". In: *Information and Software Technology* 51.2 (Feb. 2009), pp. 284–292. DOI: 10.1016/j.infsof.2008.02.004.

[Tal13] D. Talia. "Workflow Systems for Science Concepts and Tools". In: *ISRN Software Engineering* 2013 (2013), p. 15.

[Tav11] Taverna. *Taverna web site*. myGrid OMII-UK. 2011. URL: http://www.taverna.org.uk/.

[TS98] I. Taylor and B. Schutz. "Triana - A Quicklook Data Analysis System for Gravitational Wave Detectors". In: *Second Workshop on Gravitational Wave Data Analysis*. Editions Frontières, 1998, pp. 229–237.

[Tay+03] I. Taylor, M. Shields, I. Wang, and O. Rana. "Triana Applications within Grid Computing and Peer to Peer Environments". In: *Journal of Grid Computing* 1.2 (2003), pp. 199–217. DOI: 10.1023/B:GRID.0000024074.63139.ce.

[Tay+07] I. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. London: Springer, 2007. DOI: 10.1007/978-1-84628-757-2.

[TTM05] D. Thain, T. Tannenbaum, and L. Miron. "Distributed Computing in Practice: The Condor Experience". In: *Concurrency and Computation: Practice and Experience* 17.2-4 (2005), pp. 323–356. DOI: 10.1002/cpe.v17:2/4.

[Thr+10] A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. "Taming Complex Bioinformatics Workflows with Weaver, Makeflow, and Starch". In: *5th Workshop on Workflows in Support of Large-Scale Science (WORKS, 2010)*. 2010, pp. 1 –6. DOI: 10.1109/WORKS.2010.5671858.

[TC09] R. Tolosana-Calasanz. "Fostering Flexible and Dynamic Management of Scientific Workflows with Reference Nets". PhD thesis. Universidad de Zaragoza, 2009.

[TCBR11] R. Tolosana-Calasanz, J. A. Bañares, and O. F. Rana. "Dynamic Workflow Adaptation over Adaptive Infrastructures". In: *Agent and Multi-Agent Systems: Technologies and Applications*. Ed. by J. O'Shea, N. Nguyen, K. Crockett, R. Howlett, and L. Jain. Vol. 6682. Springer Berlin Heidelberg, 2011, pp. 661–670. DOI: 10.1007/978-3-642-22000-5_68.

[TC+10] R. Tolosana-Calasanz, J. A. Banares, O. F. Rana, P. Alvarez, J. Ezpeleta, and A. Hoheisel. "Adaptive Exception Handling for Scientific Workflows". In: *Concurrency and Computation: Practice and Experience* 22.5 (2010), pp. 617–642. DOI: 10.1002/cpe.v22:5.

[Tri11] Triana. *Triana project web site*. Cardiff University. 2011. URL: http://www.trianacode.org/.

[Tri15] Triana. *Triana Download*. Cardiff University. 2015. URL: http://www.trianacode.org/downloads.php.

[Tsa+04] W. Tsai, W. Song, R. Paul, Z. Cao, and H. Huang. "Services-oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing". In: *COMPSAC*. Vol. 1. 2004, pp. 554–559. DOI: 10.1109/CMPSAC.2004.1342894.

[TS12] T. Tschager and H. A. Schmidt. "DAGwoman: Enabling DAGMan-like Workflows on Non-Condor Platforms". In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. SWEET '12. New York, NY, USA: ACM, 2012, 3:1–3:6. DOI: 10.1145/2443416.2443419.

[Ull75] J. Ullman. "NP-complete Scheduling Problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384 –393. DOI: 10.1016/S0022-0000(75)80008-0.

[Uni12] T. UniProt. *UniProt: a hub for protein information*. The UniProt Consortium. 2012. URL: http://www.uniprot.org/help/uniparc.

[Vah+13] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman. "Rethinking Data Management for Big Data Scientific Workflows". In: *IEEE International Conference on Big Data*. 2013, pp. 27–35. DOI: 10.1109/BigData.2013.6691724.

[Vaq+12] L. M. Vaquero, D. Morán, F. Galán, and J. M. Alcaraz-Calero. "Towards Runtime Reconfiguration of Application Control Policies in the Cloud". In: *Journal of Network and Systems Management* 20 (2012), pp. 489–512.

[VS99] J. Vetter and K. Schwan. "Techniques for High-performance Computational Steering". In: *Concurrency, IEEE* 7.4 (1999), pp. 63–74. DOI: 10.1109/4434.806980.

[W3C15] W3C. *XML Schema Definition Language: W3C XML Schema Working Group and Schema Specifications*. W3C. 2015. URL: https://www.w3.org/XML/Schema.

[Wal+00] D. Walker, M. Li, O. Rana, M. Shields, and Y. Huang. "The Software Architecture of a Distributed Problem-solving Environment". In: *Concurrency and Computation: Practice and Experience* 12.15 (2000), 1455–1480. DOI: 10.1002/1096-9128(20001225)12:15.

[Wan+10] F. Wang, H. Deng, L. Guo, and K. Ji. "A Survey on Scientific-Workflow Techniques for E-science in Astronomy". In: *International Forum on Information Technology and Applications (IFITA)*. Vol. 1. 2010, pp. 417–420. DOI: 10.1109/IFITA.2010.210.

[WRR08] B. Weber, M. Reichert, and S. Rinderle. "Change Patterns and Change Support Features - Enhancing Flexibility in Process-aware Information Systems". In: *Data and Knowledge Engineering* 66.3 (Sept. 2008), pp. 438–466. DOI: 10.1016/j.datak.2008.05.001.

[WfM11] WfMC. *Workflow Management Coalition (WfMC)*. 2011. URL: http://www.wfmc.org/.

[WA02] M. Widenius and D. Axmark. *Mysql Reference Manual*. Ed. by P. DuBois. 1st. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN: 0596002653.

[WPF05] M. Wieczorek, R. Prodan, and T. Fahringer. "Scheduling of Scientific Workflows in the ASKALON Grid Environment". In: *SIGMOD Rec.* 34.3 (Sept. 2005), pp. 56–62. DOI: 10.1145/1084805.1084816.

[Wol+13] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. "The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud". In: *Nucleic Acids Research* (2013). DOI: 10.1093/nar/gkt328.

[Xia+12] L. Xiaorong, R. Calheiros, L. Sifei, W. Long, H. Palit, Z. Qin, and R. Buyya. "Design and Development of an Adaptive Workflow-Enabled Spatial Temporal Analytics Framework". In: *IEEE International Conference on Parallel and Distributed Systems*. 2012, pp. 862–867. DOI: 10.1109/ICPADS.2012.141.

[YB04] J. Yu and R. Buyya. "A Novel Architecture for Realizing Grid Workflow using Tuple Spaces". In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004, pp. 119–128. DOI: 10.1109/GRID.2004.3.

[YB05] J. Yu and R. Buyya. "A Taxonomy of Workflow Management Systems for Grid Computing". In: *Journal of Grid Computing* 34.3 (2005), pp. 171–200. DOI: 10.1145/1084805.1084814.

[YB10] J. Yu and R. Buyya. "Gridbus Workflow Enactment Engine". In: *Grid Computing: Infrastructure, Services, and Applications*. Ed. by L. Wang, W. Jie, and J. Chen. CRC Press, 2010. Chap. 5, pp. 119–146.

[Zha+12] J. Zhao, J. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble. "Why Workflows Break - Understanding and Combating Decay in Taverna Workflows". In: *IEEE 8th International Conference on E-Science (e-Science, 2012)*. 2012, pp. 1–9. DOI: 10.1109/eScience.2012.6404482.

[Zho92] S. Zhou. "LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems". In: *Proceedings of Workshop on Cluster Computing*. 1992.