Leandro Miguel Ribeiro Galrinho

MSc Student in Computer Science and Informatics Engineering

Live Graph Databases Using DCR Graphs

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Costa Seco, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson: Professor Maria Armanda Simenta Rodrigues Grueau
Raporteur: Professor Francisco Cipriano da Cunha Martins
Member: Professor João Ricardo Viegas da Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Live Graph Databases Using DCR Graphs

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

Nowadays, it is of uttermost importance for companies that want to be relevant on the market to produce more while making fewer mistakes. Good management practices recommend the replication of critical business operations, like hiring a new employee and the set up he has to go through to have the company tools available, or the steps and decisions required when producing some daily report. The possibility of creating and refining these processes through business process systems to better suit the daily activity of an enterprise has a direct impact on the overall productivity, organization, and cost-reduction.

The commonly used process systems make use of notations that are like state machines, having a somewhat imperative style depicting a narrow path where every decision in a process is sequential – providing the user no chance to offer input on how the process carries out – and struggle to take data into account. The proposal of several declarative languages and notations meant to solve this problem, easily incorporating data alongside the specified workflow, and providing actual control to the end-user on how the processes are accomplished by stating what can/needs to be done rather than how to do it in a step-by-step fashion.

With this dissertation we present ReDa, a novel declarative, dynamic, and reactive data-centric process language, and the mapping from its specification to a running system (the operational semantics) implemented using the mechanisms of a graph-database, namely *neo4j*. We also present and evaluate a prototype of a business process system able to emulate the process via a reactive application, addressing the challenges of having a system that interacts with a dynamic process, and the solutions adopted.

Keywords: ReDa, ReSeda, Business Process Management, Business Process Management Systems, *neo4j*, Business Engines, Graph Database, Dynamic Condition Response Graph

Resumo

Atualmente, para que uma empresa possa ser relevante no mercado é bastante importante que a sua produção aumente e que a sua taxa de erros diminua. Regras de boa prática no que toca ao controlo de processos de uma empresa recomendam que as suas tarefas mais críticas sejam efetuadas da mesma forma independentemente de quem as executa, tal como a contratação de um novo empregado e todos os passos que ele precisa de executar para que reúna as condições necessárias para trabalhar, ou quais os pontos-chave obrigatórios a seguir quando se submetem relatórios. A possibilidade de criar e ajustar estes processos ao dia a dia de uma empresa tem um impacto direto na sua produtividade, organização e redução de custos.

Os sistemas de processos mais utilizados adotam notações semelhantes a máquinas de estado, onde definem as suas atividades de uma forma sequencial e têm dificuldade em incorporar dados no processo. A proposta de várias linguagens de processos declarativas tem como objetivo solucionar este problema, permitindo a definição do processo e dos seus dados de forma simultânea e flexível, pois ao invés de se definir uma sequência de execução é possível estabelecer o que pode/tem de ser feito.

Com esta dissertação apresentamos a ReDa, uma nova linguagem declarativa, dinâmica e reativa centrada em dados, e um mapeamento desta especificação para um sistema de execução que utiliza os mecanismos de uma base de dados de grafos, nomeadamente o *neo4j*. Apresentamos e avaliamos também um protótipo de sistema de gestão de processos capaz de emular processos ReDa através de uma aplicação reativa, abordando os desafios de desenvolver um sistema que interaja com um processo dinâmico e as soluções adotadas.

Palavras-chave: ReDa, ReSeda, Business Process Management, Sistemas de Gestão de Processos, *neo4j*, Business Engines, Bases de Dados de Grafos, Grafos DCR

Contents

List of Figures

CHAPTER 1

Introduction

Companies need to establish methods that allow the quick adaptation of their activities to the newest market conditions – as they are constantly changing – and to improve pre-existing ones to accommodate customer needs. Furthermore, core tasks in a company should be executed in the same manner regardless of the employee that is working on it, and as such, should be documented. This documentation can cause a great amount of work, depending on how long and critical the tasks are, and if the market shifts in a certain direction they do not cover, either they stay obsolete, or all the documentation has to be redone. This comes down to the most simple – yet troublesome – resource of every company: time, and how much is it worth.

1.1 Motivation

To increase productivity and automate recurrent tasks, business process management (BPM) is widespread in enterprise companies. According to Palmer et al. [36], "BPM is not a piece of software, but a way of thinking and practice to optimize a company's workflow, which is supposed to serve as a bridge between people and system, being more reliable and understandable". For any number of reasons the processes inside a company may be outdated, redundant, or inefficient, and by using BPM, said processes go through detailed scrutiny while being modeled, being tuned to achieve maximum performance allowing not only time-saving by avoiding redundant tasks, but also cost reduction as productivity increases.

With this in mind, several languages and notations were proposed to define processes in information systems, and some of those are even considered a standard nowadays, such as the BPMn 2.0 notation that is discussed in Section 2.1.2. They can also be seen as communication bridges, as the steps that could otherwise be misleading are now expressed

in a standardized representation.

Process automation syncs people and system operations, and efforts are still being made to develop and improve software capable of emulating process models, providing real-time feedback on tasks deployed via monitoring features. Business process modeling systems like Camunda (Section 2.2.2) and jBPM (Section 2.2.1) achieve this by allowing the user to create, edit, and collect their analyzed model data. However, major business systems like these are based – or even exclusively support – BPMn notation, along with languages derived from it. This form of representation does not fully support all processes, not having the ideal means to model many-to-many or one-to-many relationships between activities, and not fully taking data into account, as it is detailed in Section 2.1.2.

The aforementioned motivated the proposal of several declarative languages and notations. Whilst BPMn and other similar languages specify a workflow using an imperative style, stating what has to be done step-by-step in a sequential manner, declarative languages allow the flow control to be shifted from the system to the user, by stating what can be done and letting the user choose the best course to get it accomplished. Notations like the DCR (Section 2.1.3.2) or OCBC (Section 2.1.3.3) assimilate this and take advantage of behavioral mechanisms to specify how instances of events are related amongst them. If used in a business process system, this kind of language would be a step forward to achieve more flexibility and in shifting from a "robot" paradigm, where process executioners have little or no relevance at all on how the task is going to be carried out, to one where they actually provide a meaningful contribution.

1.2 Context

In 2018, Costa Seco et al. [3] proposed a novel declarative process language ReSeDa (Section 2.1.3.4), standing for REactive SEmi-structured DAta language. It was obtained by generalizing the reactive logic inherent in spreadsheets, and extending the DCR graph process notation with it (Section 2.1.3.2). A good approach as it was, it still lacked the flexibility to work with all kinds of data and not only the semi-structured one, and hence the idea of ReDa came forth: a generalization of ReSeDa that extended its core semantics and syntax to a graph-like representation of data. Furthermore, a system that could emulate said processes and enable their interaction with the user would allow to formulate case studies and evaluate its real applicability in a business company context – ergo the proposal of this dissertation, co-authored with the dissertation advisor and researchers from the University of Copenhagen and DCR Solutions[1].

[1] https://dcrsolutions.net/

1.3 Contributions

Our contributions with this dissertation are as follows:

- We developed ReDa: a novel declarative business process language. It is a generalization of ReSeDa allowing the direct mapping from its processes to a graph-based structure. This process language is described in an accepted paper presented at DEC2H-2020, an international workshop on declarative, decision, and hybrid approaches to processes [10].
- We developed a translation tool capable of mapping the grammar of ReDa into a set of *cypher* instructions (the graph-database *neo4j*'s native language), allowing ReDa processes to be instantiated into *neo4j* maintaining all its behavior and semantics in the form of database triggers. Furthermore, case studies were also taken into account to measure the applicability of this solution.
- We developed a system that acts as the engine of a business process. It communicates with the database that has the instantiated ReDa process and supports the interaction with the user providing a correct view of its possible and mandatory actions.
- We exported the reactive components used in the system's engine to an online component collaboration library to ease further development, as each component is individually documented and maintained.

1.4 Document Structure

This current chapter provides introductory concepts and introduces context and motivation for this dissertation, why it is interesting and relevant in current days, and its novel characteristics. Next, on Chapter 2 we study the conventional process languages and notations, why they became a standard, and their possible pitfalls. Some of the most popular business management systems using these conventional languages are studied as well, together with an alternative paradigm of declarative process languages, and their advantages when compared with the traditional sequential ones.

The business process language ReDa developed in this dissertation is presented at Chapter 3, providing a formal approach to its syntax and semantics, as well as a practical example of its application. The next chapter establishes a translation procedure to map ReDa processes into a set of instructions to be instantiated into a *neo4j* graph-database (Chapter 4). The architecture of the proposed system is presented in Chapter 5, as well as the technologies used to develop it.

The practical demonstration of our prototype of a business process systems is detailed on Chapter 6, as well as some performance evaluation of case studies. At last, Chapter 7 summarizes our contributions and proposes the work that can follow from them.

CHAPTER 2

Related Work

A significant amount of work about business processes, process languages with different areas of application and its execution has been done in the past years and is still the target of a lot of scrutiny and research in academia and enterprise companies alike [11]. Since the 90's there has been a real need for building software that allows the specification of the steps of a task, their dependencies, conditions, and how these tasks are related so that a group of users can accomplish a predetermined goal [32]. This led to a big rise in the number of companies that tried to build workflow languages and systems capable of translating these languages to workflow execution processes whilst maintaining its semantics. This chapter focuses on the work previously done and how it can be crucial to determine potential pitfalls and benefits of the ReDa language and its prototype of a business management system.

2.1 Business Process Modeling

As captured by Palmer et al.[36], BPM can be seen as a "discipline involving any combination of modeling, automation, execution, control, measurement and optimization of business activity flows, in support of enterprise goals, spanning systems, employees, customers, and partners within and beyond the enterprise boundaries". This means that BPM is not a piece of software or technology, but a practice done by people of an organization to maximize the improvement of a process or a task, minimizing its cost.

Quality in organizations can thereby be seen as a consequence of using process models, and many efforts are still being made to build enterprise-grade applications that incorporate them. They help in the organization and assignment of tasks by standardizing them across an enterprise so they can be easily understood and maintained, therefore mitigating human errors [28]. This efforts may take the form of languages like BPMl (Section 2.1.1), OCBC

(Section 2.1.3.3), BPMn (Section 2.1.2), DCR (Section 2.1.3.2) ReSeDa (Section 2.1.3.4) or even ReDa (Chapter 3), which are discussed in their respective sections. They may as well be associated with engines (discussed on Section 2.2) belonging to full fledged business process system applications.

In this section, the same example of a process with its behavior and data associated is modeled using distinct languages and notations, evidencing the inherent advantages of using a declarative approach rather than the strict sequential approach used on imperative languages. The example consists of a veterinary clinic, where clients can create an appointment for one or more pets with a veterinarian, having an option to fill a form for each, stating its purpose as a description, and to perform the check-in of them all. Note that one appointment may consist of several pets – having several forms – but each form corresponds to only one appointment. To conclude, each of the check-ins corresponds to one and only one check-out, as all pets are picked up by their respective owner at the same time.

2.1.1 BPMl

The business process modeling language specification (BPMl) was introduced in 2000 as a standard to specify business process management [18], with the objective of being a specification language capable of defining any executable process via a business process management system. It is based on the XML language and is oriented towards execution: defines the process in a sequence of operations – atomic or not [29, 18] – performed in a certain context.

The following listing based on the veterinarian clinic example presented on Section 2.1 depicts the common BPMl structure for a simple business process, where a sequence (denoted by the tag <sequence>) consists of operations (denoted by the tag <operation>), whose attributes and child nodes include the participants and activity to be described. In this case, only the `choose_pet` operation is modeled (in a simplified manner).

```
<process name = "Veterinarian_Clinic_Example">
    <sequence>
        <operation name="Choose_Pet_Action">
            <participant name = "Pet_Owner"/>
            <output message = "Choose_Pet">
                <assign to="Action">Pet Form</assign>
            </output>
            <input message = "Pet_Name">
                <assign to="filter" from="Text" />
            </input>
        </operation>
        (...)
    </sequence>
</process>
```

As already stated, BPMl is oriented towards process specification and it is hardly readable by humans. For that reason, an effort was made to develop a visual notation that easily communicates process models: BPMn, discussed in Section 2.1.2. BPMl was later deprecated in 2008.

2.1.2 BPMn

The business process modeling notation (BPMn) was created out of the necessity to visually describe a business process. It is widely accepted in industry and academia alike, being an ISO standard in its 2.0 version [11]. It allows modeling various types of business processes and the way they relate to each other, by using a graph-oriented notation that shares some resemblance to the one used in DCR (Section 2.1.3.2). BPMn elements can be divided into the following groups:

- The activities – tasks, processes (and subprocesses that encapsulate other tasks), and others. They are represented by a rectangle with rounded corners.

(a) Call activity (b) Subprocess activity (c) Task activity

Figure 2.1: Example of some BPMn activities

- The artifacts – they do not influence execution semantics but provide an explanation to some part of the diagram or stress the information that is needed in some activity.

(a) Text annotation artifact (b) Group artifact

Figure 2.2: Example of some BPMn artifacts

- The events – start event (thin outline), intermediate event (dotted outline), and end event (thick outline). Each of these has multiple sub-events, like timed events or message events, which are represented by circles and may contain other symbols based on their type.

(a) Timed start event (b) Msg. intermediate event (c) Termination end event

Figure 2.3: Example of some BPMn events.

- The gateways represent decisions, merges, joins, amongst other things. They are represented by a diamond-shaped object, containing a symbol depending on the type of gateway.

(a) Exclusive gateway (b) Parallel gateway

Figure 2.4: Example of some BPMn gateways.

- The arrows on the diagram connecting its elements are called "connecting objects", and they may be sequence flow arrows (represented by a full arrow), message flow arrows (represented via a dashed arrow), or association arrows (dotted arrows regarding an association of data to an element, like the textual description of the reason motivating an appointment on our veterinarian clinic example, for instance).

Figure 2.5: Example of BPMn arrows.

- Pools consisting of several lanes, each one representing a major participant of the process with its tasks and how they relate to other lanes (an example is presented in the modeling of our veterinarian clinic example).

Getting back to the veterinarian clinic example, Figure 2.6 depicts the modeling of said process using a BPMn notation. There is a pool with a lane representing the pet owner, and the process starts with him creating an appointment. Next, the owner chooses a pet and fills out the form for him, giving the appointment description as optional information, repeating this process for each of the pets. When finished, the owner checks-in the appointment and picks his pets once the appointment is concluded.

Whilst the modeling of this process is possible using BPMn, it is necessary to resort to artifacts to incorporate data into it, and the cardinality constraints between activities are unclear – it is not obvious that there is a relation of many-to-one between the `fill_out_form` and the `check_in` activities. Furthermore, the BPMn specification is lengthy, complex, and ambiguous, making its use susceptible to interpretation problems. As pointed out in [2], this standard fails to be implementable due to the behavioral issues on its concepts, making its implementation only viable on a limited subset of the language.

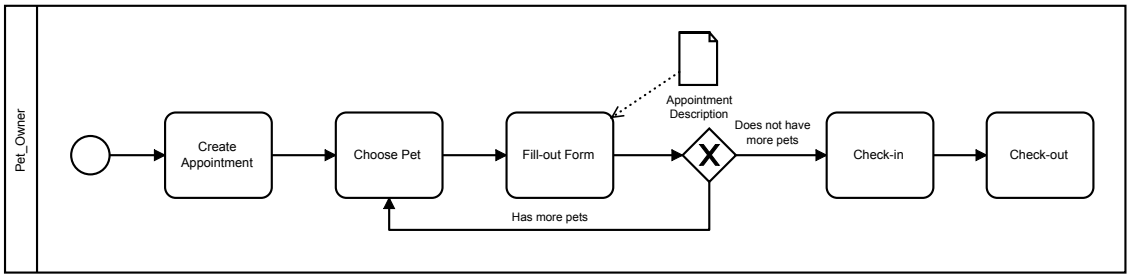

Figure 2.6: BPMn diagram of the veterinarian clinic example.

Since interpretation can be personal, and different persons can have different understandings of the same construct, the user may understand the business process described in a way that differs from what it is compiled to. For instance, Borger et al. [2] substantiated this by pointing out that the life-cycle concept together with the interruption constructs available in the BPMn 2.0 specification, like exceptions, can not specify what happens to the process if an exception is fired at a sub-process level: one compiler could be programmed to interrupt all the process at once, whereas another compiler could opt to interrupt solely the sub-process that fired the exception. The major flaws present in a list of 435 (as of November 2020) open issues – related to problems with specification ambiguity – listed at the BPMn 2.0 official page [35] are presented next:

- Concepts are semantically underspecified leaving a window for misinterpretations;
- Poor conceptual support for numerous features relevant in the design of business processes;
- Complex description of concepts and many of them are defined in terms of other constructors, forcing the reader to simultaneously and repeatedly consider multiple sections of the standard document;
- There is no systematic mechanism for refining a model from conceptualization to its execution.

This makes the communication between different systems and the portability of models (almost) impossible, as even a tiny bit of ambiguity can lead to big differences in the semantics of the same process. This is a major pitfall of BPMn because it fails to be what it intended – standardized – as the models are platform-dependent (some experts advocated the creation of a document providing a reference implementation where the ambiguous constructs could be clarified [2]).

Furthermore, it lacks the means to specify cardinality constraints and the interaction between instances of the same process, thus existing no way of modeling a one-to-many or many-to-many relationship on BPMn without creating multiples lanes containing the same instance of one element or other similar complex solutions, as it can only model the life-cycle of one process instance at a time [25]. According to J. Su et al [24]: “The processes serving a common business service must be interrelated by sharing data (...). The exclusion of data in these models limits applicability”. Indeed, if processes take data into account

it becomes possible to study the relationship between different business processes via the common data they share. This topic lead to the arrival of many languages and notations capable of integrating data into process modeling, such as the OCBC notation [17, 25] approached in Section 2.1.3.3, the ReSeDa language [3] discussed on Section 2.1.3.4, and the ReDa language on Chapter 3.

2.1.3 Constraint Based Models

As previously stated, conventional business process models can only describe the life-cycle of one instance at a time and in isolation and struggle to include data-elements into account. For that reason, other forms of representation more convenient to specify process behavior, as well as complex interactions between different types of instances, are often used alongside them.

This is accomplished by using constraint-based models that declaratively state what can be done in a process – the decision making is shifted from the system to the user – instead of the traditional imperative approach, which sequentially specifies how the process is going to be executed, providing the users limited or almost no impact on how the process is going to be carried out [20]. The following process languages and notations discussed next provide a relevant contribution on how to incorporate more flexibility into a business process.

2.1.3.1 DECLARE

DECLARE is a predecessor of some of the declarative process languages used nowadays – like the DCR Section 2.1.3.2 – and was a prototype of a business process modeling system (ceased to exist circa 2012) using a constraint-based process modeling language [20]. It was able to support loosely-structured processes while maintaining the major benefits of the conventional business process systems that used imperative languages, such as model verification, analysis of past executions, and being able to change models at run-time.

Whereas business languages such as the BPMn (Section 2.1.2) specify step-by-step the execution process of a certain task, declarative languages bring the flexibility of specifying what has to be accomplished and letting the user choose the best course to get it done. DECLARE was developed as a declarative constraint-based system, allowing the user to customize its own relation type specification. It was possible to create many of the nowadays existing binary and unary constraints, by specifying its unique name, its semantics (using a linear temporal logic formula), and its graphical representation. For instance, it was possible to create the "response" constraint present in the DCR notation (Section 2.1.3.2), as seen in Figure 2.7.

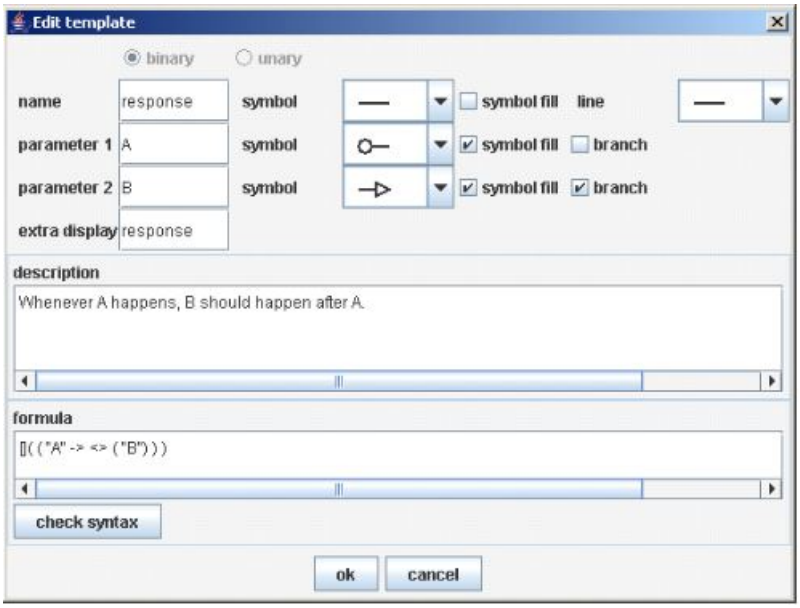

Figure 2.7: Creation of a "response" like constraint on DECLARE.

DECLARE also supported the notion of mandatory and optional constraints. The first adopted the logic from the imperative constraint models, being useful to model a critical part of the system prone to human errors. The latter allowed the user to decide whether or not to follow a specific path, providing more flexibility in an otherwise linear path.

2.1.3.2 Dynamic Condition Response (DCR) Graphs

With the growth of business process design technologies came the development of many flow-oriented process languages and notations, such as the BPMl (Section 2.1.1) and the BPMn (Section 2.1.2). Whilst the specification of how a process should behave from start to end in an imperative way (sequentially stating the execution steps) is important in a business process management system, there is also a need to identify the business and compliance rules restricting the orders between the system changes. DCR graphs focus on the logic behind the process, describing in a declarative way the causal relationships and pending obligations between the events in a system [5, 6]. This means that the exact sequence of actions is left undefined but always restricted to the set of constraints they must respect, thus giving the system maximum flexibility.

Dynamic condition response (DCR) graphs are, as the name suggests, graphs. As so, events are represented as elements and are related to each other by relations restraining their behavior and order of occurrence. Getting back to the veterinarian clinic example, its DCR representation can be seen on Figure 2.8.

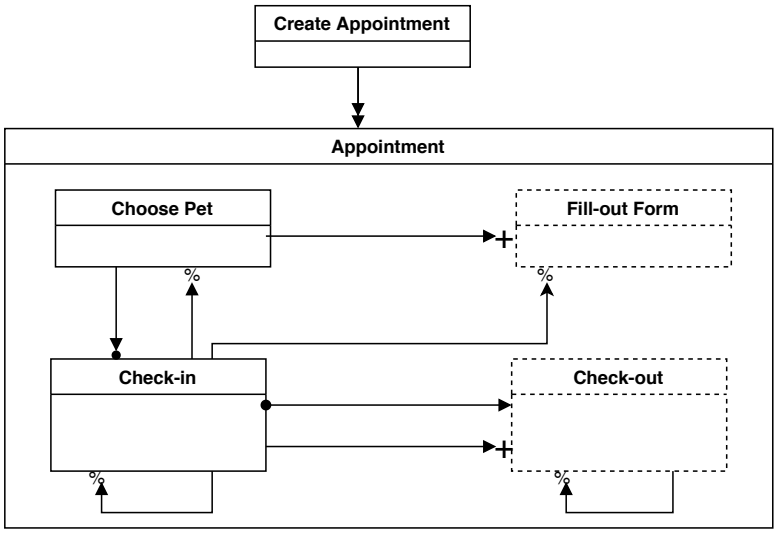

Figure 2.8: DCR modeling of the veterinarian clinic example.

To represent the state of a process, each element is associated with a set of properties – the markings. One of them is the included property, denoting if an event is available to be executed. We can see on Figure 2.8 that the `fill_out_form` and the `check_out` elements are depicted using a dotted outline, meaning they have their included property set to false, and cannot be executed unless they are explicitly included into the process. Following, the executed property conveys if a specific event has already happened, and if so, it has a value property representing its current value. Lastly, the pending property states if an event is obliged to happen eventually for the graph to reach a valid final state. A new DCR graph representing a process must have some of its events enabled, and when said events are executed, new system markings are produced [16][8].

There is a total of six different kinds of relationships between events to define their behavior (authors in [1] even added a time mechanic, proving itself very useful in some cases):

- the condition relation, represented by $e \longrightarrow\bullet\ e'$ and present between the `choose_pet` and `check_in`, meaning that the latter event can only occur when `choose_pet` has already occurred at least once;

- the response relation, represented by $e \bullet\longrightarrow e'$ and present between the `check_in` and `check_out`, meaning that the `check_out` must happen after `check_in` has been executed;

- the exclusion relation, represented by $e \longrightarrow\!\!\!\!\!\%\ e'$ and present between the `check_in` and `choose_pet` events, as well as in the unary relations that the `check_in` event and `check_out` event have, and denote that when the first happens the latter becomes excluded from the process and can no longer occur unless explicitly included again;

- the inclusion relation (the semantic opposite of the exclusion relation), represented by $e \longrightarrow\!\!+\ e'$ and present between the `check_in` and `check_out` events, meaning that when the first happens the latter becomes included in the process and can occur again (if previously excluded);

- the fifth relation in DCR is the milestone relation represented by $e \longrightarrow\!\!\diamond\ e'$, with a semantics close to the condition relation, where e needs to be excluded or not pending for e' to be executed; and finally,

- Debois et al. [5] pointed out that the DCR graph notation was conceived as both generalization of event structures and a generalization of the process matrix. This notation kept being developed and a notion of "dynamic creation" of a sub-process was found lacking. Hence, authors in [6] came up with this novel sixth relation that specifies a spawn event: when a reproductive event ***e*** happens in an instance of a process T, a copy of T (T') is created and processed in parallel, and as the process evolves both T and T' can be updated. In the veterinarian clinic example, this relation is present between the `create_appointment` and the `appointment` elements, meaning that whenever `create_appointment` is executed a new sub-process with an `appointment` is created, and both can be executed in parallel.

A valid DCR graph must have some of its elements enabled for execution – at least one event must be included and have no condition or milestone relation preventing its execution. On the veterinarian clinic process, in its initial state, only the element `create_appointment` is available as all the other elements are modeled as being part of the sub-process created when the former is executed. This event (`create_appointment`) is included and has no inward condition relation at all, so it is a valid element to execute. When doing so, the remaining four elements are incorporated into the process – the `fill_out_form` and `check_out` start initially excluded and cannot be executed, and the `check_in` event has an inward condition relation where `choose_pet` has not yet been executed, and as such, in this sub-process level, the only action possible is to choose a pet to schedule an appointment to.

When executing the `choose_pet` activity two things happen: the `fill_out_form` becomes included and enabled, and the `check_in` becomes enabled as well, as its outward condition relation element has already been executed. At this point in the process, there are three actions possible: to check-in the appointment with only one pet chosen and no form given, to choose more pets and do not fill any form, and to fill as many forms as pets chosen before checking in. When checking in, the `choose_pet`, `fill_out_form`, and `check_in` activities become excluded from the process, and the `check_out` becomes included and pending, meaning it has to happen eventually for the process to reach a valid final state of execution. When executing the `check-out` event, this activity also excludes itself from the process and this sub-process reaches an end, as there are no more activities susceptible to execution.

The DCR way of declaratively modeling a process provides much more flexibility than the previously seen BPMn model, as the constraints present on the graph do not restrict the order of execution of events, shifting the decision of which to perform to the user.

2.1.3.3 OCBC: Object-Centric Behavioral Constraint Model

In previous existing approaches (such as the widely used BPMn) there is a clear separation between behavior and data flow in process models. That is why a declarative approach that can describe processes with interacting instances and data dependencies provides substantial applicability in process modeling. The Object-Centric Behavioral Constraint model and notation introduced in [17, 25] is an approach to this problem, by using cardinality constraints – as the ones present on Figure 2.9 – capable of specifying both the structure and the behavior of a program in a single diagram, where different kinds of instances can interact between them while taking data into account.

- Unary response – if left executes, then right must execute exactly once afterwards;
- Unary precedence – if left executes, then right must have been executed exactly once;
- Response – if left executes, then right must execute afterwards;
- Precedence – if left executes, then right must have been already executed;
- Non-response – if left executes, then right does not execute again;
- Non-precedence – if left executes, then right was never executed;
- Non-coexistence – left and right cannot be both executed; and finally,
- Coexistence – if left executes, then right must have been executed before or after.

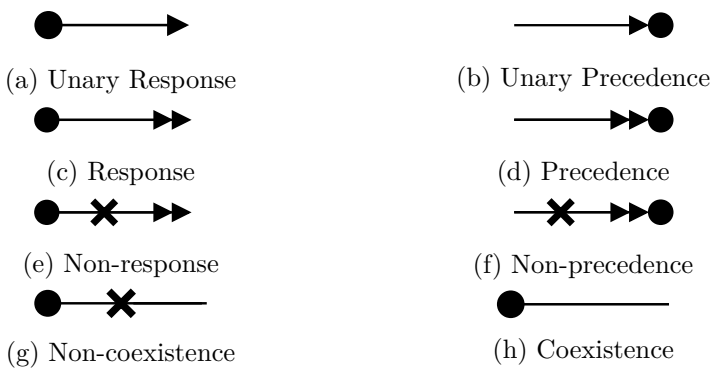

Figure 2.9: The OCBC model relations

The OCBC model of the veterinarian clinic, as well as the logic behind it, are presented next. The diagram of figure Figure 2.10 specifies that: `(1)` each appointment is for at least one pet; `(2)` a pet can only be chosen after the appointment is created; `(3)` to each pet there is a form that can be filled; `(4)` each form is submitted to only one check-in; `(5)` a check-in may have several forms submitted; and finally, `(6)` there is only one check-in and one check-out. This can be easily modeled by the OCBC notation, however, this notation is not directly executable as it is not supported by any business process management system

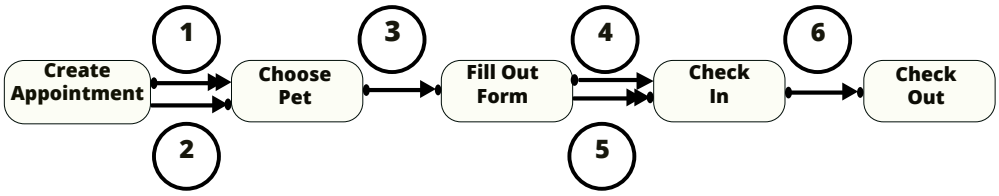

Figure 2.10: OCBC model of the veterinarian clinic example

.

2.1.3.4 ReSeDa

ReSeDa stands for REactive SEmi-structured DAta and it is a declarative data-centric novel approach proposed by the authors in [3] for a process language. It generalizes and was inspired by both the DCR notation (Section 2.1.3.2) and the reactive properties of widely used spreadsheets [21]. As such, a ReSeDa element can be either an:

- Input data-element, meaning that it expects a value to be inserted by the user that triggers an event and assigns its value to the corresponding data field;
- Computational data-element, that is computed automatically (as done in spreadsheets).

The semantics of ReSeDa is based on the semantics of DCR, adopting its notion of markings and the relationships between elements. Each element has a value, can be included (excluded), pending (not pending), and executed (not executed). For the program to reach an acceptable state, there should not exist any pending data-elements.

These elements are also restricted by the six constraints relations discussed in Section 2.1.3.2, giving them the semantic foundation they need to specify its relative order and response obligations, as well as the dynamic creation of new sub-processes. It is also possible to access specific data-elements or refer to other events by their path expression (their location). The veterinarian clinic example is modeled next using the ReSeDa process language. It is also given a lightweight approach to its semantics and syntax, which are formally defined in the paper that introduced it [3]. The example is going to have three collections of data-elements, each establishing its own scope: the client collection, the pet

collection, and the veterinarian collection, each of these can have nested data-elements modeled in the following manner:

```
1  clients[]{
2    create_client[?:string]
3    ;
4    create_client -->> {
5      client[]{
6        client_id[freshid()],
7        client_name[@trigger:value]
8      }
9    }
10 }
```

The `clients` data-element collection has only one input data-element in line 2 (named `create_client`) with type string as an expected input value. When executed, the spawn event in line 4 creates a sub-process (identified by being enclosed in curly braces) representing a client belonging to the collection `clients`, and the data-element `client_name` will have as value the string given as argument in the data-element that spawned it (specified by the `@trigger:value`) – in this case, the data-element `create_client` in line 2. Thus, this data-element is a rule of the process that creates a new data-element `client` in the scope of the collection of data-elements `clients` when executed. The other data-element named `client_id` has its value auto-computed and predefined by the function `freshid()`, providing a distinct id for each client. The remaining veterinary and pet collections are very similar to the one just described.

To execute a data-element we need to access it providing its path and a value as argument. For example, on the pet collection there could be an input data-element `petName`, and for the user to provide its value it would have to access it providing: `pets/pet[0]/petName` with "Pantufas" as argument, meaning that in the collection of `pets` it is selected the data-element `pet` with the id zero, and its `petName` is modified to "Pantufas".

Focusing on the workflow, if a client wants to book an appointment he needs to create it first (`create_appointment`) providing its `client_id` and the `vet_id`. After this, it is possible to choose the pet(s) for said appointment (`choose_pet`), and fill-out the form for each one of them, optionally providing the reason to book the appointment as a description. The next step is to `check_in` the appointment followed by its respective `check_out`. This sequence of actions, with the addition of constraints from `1` to `6`, are modeled using the ReSeDa notation in Figure 2.11.

- `(1)` each `create_appointment` event has at least one `choose_pet` event;
- `(2)` each `choose_pet` event is preceded by a `create_appointment` event;
- `(3)` to each `choose_pet` event corresponds one and only one `fill_out_form` event;
- `(4)` the `fill_out_form` event has only one `check_in` event that follows it;

- `(5)` each `check_in` event can be preceded by multiple `fill_out_form` events;
- `(6)` to each `check_in` event corresponds one and only one `check_out` event.

The collection of `appointment` data-elements (line one of Figure 2.11) encapsulates all the logic previously described. The semicolon in line three separates data-elements from the rules of the process – the relationships in lines twenty-one to twenty-six, defining the behavior between data-elements just like on the DCR graph notation.

In the first place, `(1)` states that each `create_appointment` data-element has at least one `choose_pet` data-element: this is handled at line nine as the `choose_pet` data-element is initially not executed (it has no `:` behind it), and is pending (it has `!` as a prefix), meaning it has to be eventually executed, and as there are no other rules that mention this data-element, it can happen any number of times.

Next, `(2)` requires each `choose_pet` data-element to be preceded by the data-element `create_appointment`. This is also handled in line nine as this data-element belongs to the sub-process that spawns with the `create_appointment` data-element, and is only able to be executed once the latter happens. Also, the condition rule in line twenty-one states that only when a `client_id` is provided (which is when an appointment is created), the `choose_pet` data field becomes enabled and can be executed.

The third rule states that to each `choose_pet` data-element corresponds one and only one `fill_out_form` data-element. This is handled in lines thirteen and fourteen as each time a pet is chosen a new sub-process to fill out the form is spawned, and the pending data-element `pet_id` from line sixteen has the id from the pet chosen.

Next, `(4)` requires that all the forms of an appointment are preceded by one and only one `check_in` data-element: this is accomplished by excluding the `choose_pet` data-element after executing the `check_in` data-element, as stated in line twenty-four. Once this happens it excludes itself, as stated in line twenty-three, ensuring that this data-element can only happen once. The fifth constraint specifies that each `check_in` data-element can be preceded by multiple `fill_out_form` data-element, and this is obtained by allowing multiple executions to the `choose_pet` data-element (as each one has its own form), and they are all checked-in at the same time.

Lastly, `(6)` states that to each `check_in` data-element corresponds one and only one `check_out` data-element. As already mentioned, it is only possible to execute the `check_in` data-element once, as it excludes itself and the `choose_pet` data-element, which was the only data-element that could re-include it. By saying that each `check_in` data-element includes a `check_out` data-element, and each of the latter once executed also excludes itself, as present at line twenty-six, `(6)` is guaranteed.

Semantically speaking, ReSeDa is defined by a labeled transition system where states represent the data-elements, with their respective marking information, and the transitions correspond to the execution of said data-element, modifying their markings.

```
appointments[]{
  create_appointment[?:@/clients/client/client_id:value]
  ;
  create_appointment -->>{
    appointment[]{
      !appointment_id[freshid()],
      !client_id[@trigger:value],
      !vet_id[?:@/vets/vet/vet_id:value],
      choose_pet[?:@/pets/pet/pet_id:value],
      check_in[?:true],
      %check_out[?:true]
      ;
      choose_pet -->>{
        form[]{
          !form_id[freshid()],
          !pet_id[@trigger:value],
          description[?:string]
          ;
        }
      },
      client_id -->* choose_pet,
      choose_pet -->* check_in,
      check_in  -->% check_in,
      check_in  -->% choose_pet,
      check_in  -->+ check_out,
      check_out -->% check_out
    }
  }
}
```

Figure 2.11: ReSeDa modeling of the veterinarian clinic example.

2.2 Business Process Modeling Engines

With the increasing need for enterprise-grade quality of service requirements such as automation and scalability in company applications [23], most of them include a way to maximize this in the form of a workflow management component. This piece of software may have many designations, such as Business Process Modeling Systems (BPMs) or Workflow and Decision Automation Platforms [30], and can be implemented in different languages and offer distinct functionalities, but at its core, they all share the same vital piece of software responsible for the processing and monitoring of business processes: the business process engine.

This engine may as well be implemented with different architectures: authors from [13] discuss the perks of having a distributed engine running on a server instead of a local one, arguing that the gains of autonomy and scalability are enough to out-value the costs of maintaining a decentralized one. But despite its location, in a BPMs the engine has the responsibility of loading a business process, to check which tasks are enabled, to allocate the correct tasks for each user, to produce a new system state after a valid execution, and even to generate global reports with key performance indicators for company analysis.

The widely adapted notation on business process engines is the BPMn, however, as there is no certification authority to check their standard conformance to the BPMn implementation [11], and even though the direct execution of a process model enabled by its 2.0 version meant to minimize the gap between desired and actual behavior, various studies [11, 12, 23] prove that some BPMn features are almost never implemented, or they are but in a differing way (as previously discussed on Section 2.1.2), and thus only a small dialect of the language is common in the majority of engines, defeating the purpose of standardization. On the following subsections are presented two major open-source BPMs's and how they implement processes as specification and its respective monitoring.

2.2.1 jBPM

jBPM is an open-source BPM engine written in Java that supports BPMn 2.0 specification, and its currently at its 7.31.0.Final version. It can be used in a Java environment where the engine is embedded in the application, or as a standalone service [31], where it is deployed on the cloud. As any BPMs its intention is to shorten the gap between a business analyst and the developer, and it accomplishes this by providing a simple graphical interface where it is possible to specify a process by drag-and-drop of BPMn components, generating the Java code semantically equivalent to it, remaining fix once deployed (Figure 2.12).

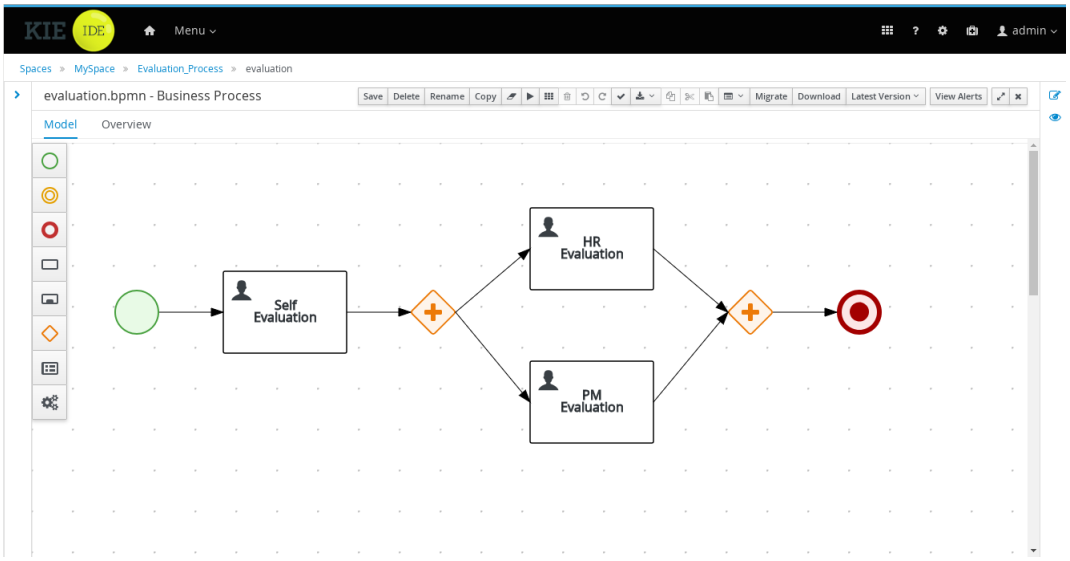

Figure 2.12: jBPM web-based graphical user interface, present in [31].

If used in a Java environment like Eclipse, at first a plug-in has to be installed [1]. From there on, when creating a new jBPM project (or using it as a dependency in a Maven project) a screen similar to the presented in Figure 2.13 shows up, and the Java code specification of the desired BPMn process and its visual representation is visible, via an interface on top of it.

[1] Available at http://downloads.jboss.org/jbpm/release/6.0.1.Final/updatesite

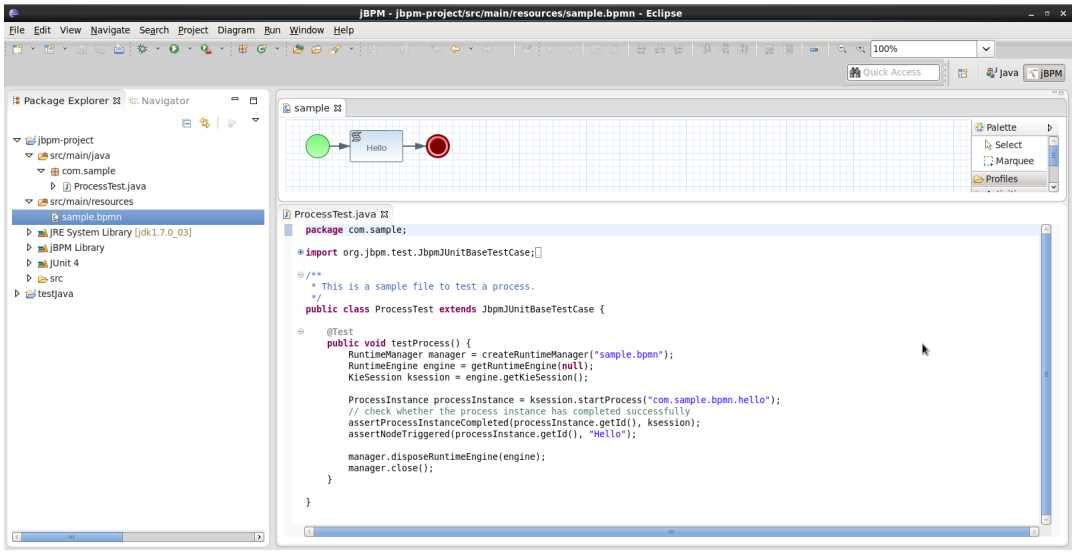

Figure 2.13: jBPM on eclipse via plugin, present in [31].

It also comes with a module for process monitoring (Dashbuilder) allowing data of heterogeneous processes to be displayed with metrics and performance indicators. This module is connected to the engine and fetches information via SQL queries related to the data currently shown on the dashboard (Figure 2.14).

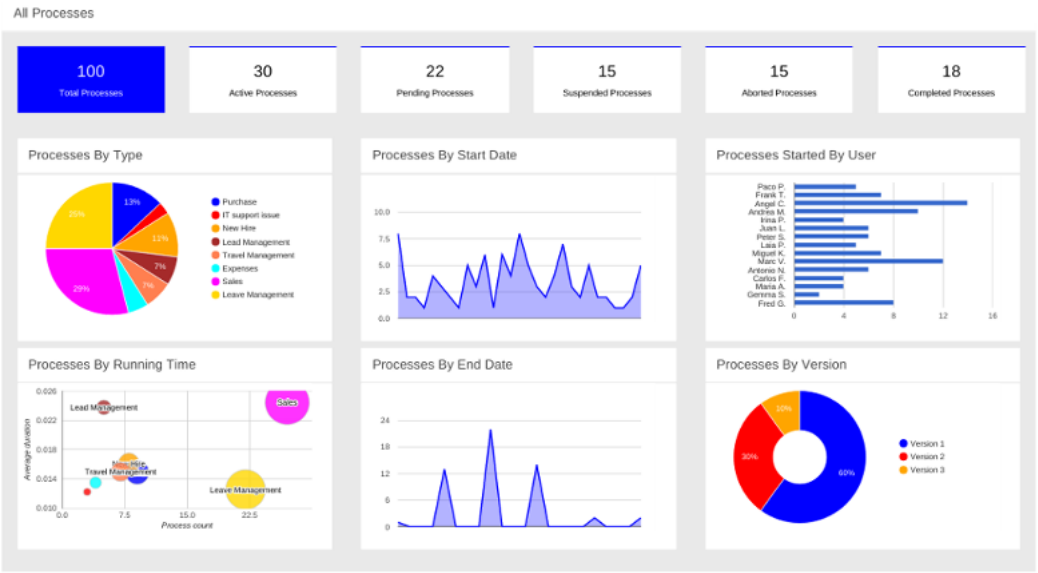

Figure 2.14: *jBPM* monitoring dashboard module in [31]

2.2.2 Camunda

Together with jBPM, Camunda is one of the most used and complete open-source BPMs on the market [30]. It is also a framework built on Java that supports BPMn 2.0 specification, and offers support for semi-structured data such as XML or JSON. It is currently on version 7.12.

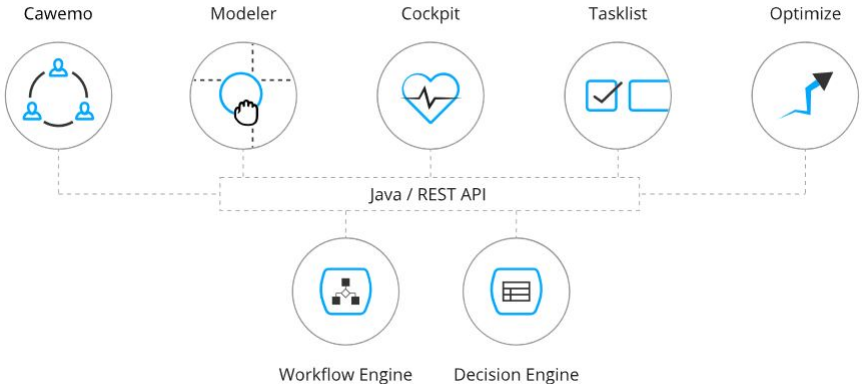

Figure 2.15: Camunda overview [30].

As depicted in Figure 2.16, Camunda offers an application similar to jBPM where it is possible for a developer to specify a business process by drag-and-drop of BPMn 2.0 components, which is translated to Java code once deployed (the Modeler). For example, a portion of the process modeled in Figure 2.16 is represented in Figure 2.17. Furthermore, the Camunda Cockpit – a web application much like the one of jBPM depicted in Figure 2.14 – provides monitoring features for the information related to deployed business processes. Aside from these two components and as depicted in Figure 2.15, Camunda also provides a range of other collaborative applications that interact with the engine via a REST API, that help to specify and study the processes to be deployed (Cawemo), a service where each user can see the tasks assigned to them (Tasklist), and a feature that allows the creation of reports referring to deployed processes (Optimize).

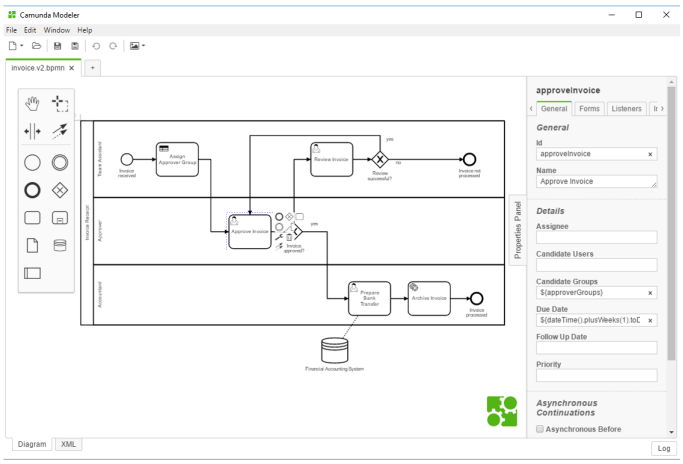

Figure 2.16: Camunda graphical interface [30].

```
BpmnModelInstance modelInstance = Bpmn.createProcess()
  .name("BPMN␣API␣Invoice␣Process")
  .[...]
  .exclusiveGateway()
    .name("Invoice␣approved?")
    .gatewayDirection(GatewayDirection.Diverging)
  .condition("yes", "${approved}")
  .userTask()
    .name("Prepare␣Bank␣Transfer")
    .camundaCandidateGroups("accounting")
  .serviceTask()
    .name("Archive␣Invoice")
    .camundaClass("org.camunda.bpm.example.invoice.service.ArchiveInvoiceService")
  .endEvent()
    .name("Invoice␣processed")
  .moveToLastGateway()
  .condition("no", "${!approved}")
  .userTask()
    .name("Review␣Invoice")
    .camundaAssignee("demo")
  .[...]
  .done();
```

Figure 2.17: Abbreviated Java code representing the BPMn process on Figure 2.16

Chapter 3

ReDa - Reactive Data-driven Processes

Current business process modeling technologies based on flow-oriented process notations, such as the already discussed BPMn (Section 2.1.2), have some kind of support to incorporate data into the process – by referral to a specific document or database to assist in the process decision making. However, as data is mainly handled outside the process control, this hinders representation and reasoning about the full process behavior [4].

This chapter presents ReDa – a declarative and data-driven generalization of ReSeDa. It focuses on a native graph-based representation consisting of data items and relationships between them, where constraints and computations can be controlled dynamically according to the current state of the process, and allowing elements to be queried in a graph-based natural way by traversing the data. ReDa may as well be seen as a "graph spreadsheet" where instead of a matrix elements are nodes in a graph, and the usual spreadsheet updates are external executions that may lead to recompute the value of dependent nodes, thus providing a reactive behavior to the language much akin to the one seen at spreadsheets. Additionally, the DCR constraints between elements adopted by ReDa provide more control than the present at spreadsheets.

3.1 Syntax

The formal syntax of ReDa is depicted in the grammar in Figure 3.1, and the intended semantics for each construct is explained on Section 3.2. The syntax closely follows that of ReSeDa [3], with obvious differences in the query language for data-elements and the addition of explicit data relationships, allowing the selection of data-elements that satisfy some criteria (related to their properties or existing relationships) and to dynamically control the process behavior.

P	::=	$\overline{D}\,;\,\overline{R}\,;\,\overline{Y}$	Processes
D	::=	$(n_\rho : \ell)[?{:}T]{:}(h,i,r,v) \mid (n_\rho : \ell)[E]{:}(h,i,r,v)$	Data-Elements
Y	::=	$\phi \dashv [E] \mapsto\!\bullet\ \phi \mid \phi \dashv [E] \mapsto\!\!\!\!\infty\ \phi \mid \phi \bullet\!\!\dashv [E] \mapsto \phi$	
	$\mid$	$\phi \dashv [E] \mapsto\!\!+\ \phi \mid \phi \dashv [E] \mapsto\!\!\%\ \phi \mid \phi \dashv [E] \mapsto\!\!\!\!\gg P$	Control Relationships
R	::=	$\phi \dashv [n : \ell] \mapsto \phi$	Data Relationships
T	::=	$\mathsf{Unit} \mid \mathsf{String} \mid \mathsf{Number} \mid \mathsf{Boolean} \mid \mathsf{List}\ T \mid \{\, \overline{x : T} \,\}$	Data Types
ϕ	::=	$\psi\ \mathsf{RETURN}\ E \mid n$	Node Queries
ψ	::=	$\mathsf{MATCH}\ Q\ \mathsf{WHERE}\ E$	
	$\mid$	$\mathsf{MATCH}\ Q\ \mathsf{WHERE}\ E\ \mathsf{WITH}\ \mathit{pipe}\ \mathsf{WHERE}\ E$	Match Expressions
E	::=	$c \mid n \mid Q \mid \phi \mid \psi \mid E{:}\mathit{attr} \mid f(E_1,...,E_n)$	
	$\mid$	$\{\, \overline{x = E} \,\} \mid E.\ell \mid [\overline{E}] \mid \mathsf{hd}(E) \mid \mathsf{tl}(E)$	Expressions
c	::=	$\mathit{numbers} \mid \mathit{strings} \mid \mathsf{true} \mid \mathsf{false} \mid \mathbf{1} \mid \bot$	Literals
pipe	::=	$\mathit{agg}\ \mathsf{AS}\ n \mid n\ \mathsf{AS}\ n$	Pipeline Term
agg	::=	$n \mid \mathsf{COUNT}(E) \mid \mathsf{MAX}(E) \mid \mathsf{MIN}(E) \mid \mathsf{SUM}(E) \mid \mathsf{AVG}(E)$	Aggregating Functions
attr	::=	$\mathit{value} \mid \mathit{executed} \mid \mathit{included} \mid \mathit{pending}$	Attributes

Figure 3.1: Syntax of ReDa

A ReDa process P comprises a sequence of data-element definitions ($\overline{D}$), followed by a sequence of definitions for data relationships ($\overline{R}$) and a sequence of control relationships ($\overline{Y}$) (the concrete syntax uses a comma as a separator in these sequences and a semicolon as a separator between collections). These definitions are based both on a query language on graphs and a language of expressions that manipulate a set of data types like unit, numbers, strings, booleans, lists, and records [10].

ReDa processes are composed of data-elements and the existing relationships between them, and they can act either as a way to interact with the process allowing the input of some required data (input data-element), or as a way to show information (output data-element). The first ($(n_\rho{:}\ell)[?{:}T]{:}(h,i,r,v)$) defines a system entry to allow the interaction between the user and the process, and the second ($(n_\rho{:}\ell)[E]{:}(h,i,r,v)$) defines an element in the graph that holds the result of a computation, given by expression E. A computational element may have data dependencies referring to values carried by other data-elements defined by the expression E, making its re-evaluation necessary whenever one of them is executed.

In their concrete syntax, both input and computational data-elements have: `(i)` a local identifier (n), used to identify the data-element in the context it was defined and its sub-contexts; `(ii)` a unique runtime identifier (ρ), only generated after the data-element has actually been created; `(iii)` a label (ℓ), that represents the type of the node (e.g. if it is an input node or a computational node, but can also be used to assign roles in a process, e.g. manager, employee...); `(iv)` either a type (T, if it is an input data-element) or an expression (E, if it is a computational data-element); and finally, `(v)` a marking (h,i,r,v), which is typical on DCR related languages [3, 7, 15], and defines the current state of the data-element on the process, being comprised of:

1. an integer value denoted by h for `happened` or `executed` (in the concrete syntax), greater than zero if the event has been previously executed (executions are external events where the user interacts with the process, and each of this interactions updates the respective data nodes `executed` property by one);

2. a boolean denoted by i for `included` (in the concrete syntax) indicating whether the data-element is currently included in the process. A data-element that is not included is considered irrelevant: it cannot execute, and cannot prevent the execution of others (an element that is "not included" is considered excluded);

3. a boolean denoted by r for `required` or `pending` (in the concrete syntax) indicating whether the data-element has a pending execution. A pending element is required to be subsequently updated (or become excluded) at some point in the future for the process to reach a valid termination; and finally,

4. a `value` denoted by v indicating the current value of the data-element if it has been executed ($h \geq 1$), or the undefined value ($\perp$) if not, meaning its initial value is dependent on external interaction with the process.

Data-element definitions in the examples ahead are simplified by not having the markings field for abbreviation purposes, and have a symbol prefixed to them to differ from the default markings (included and not pending): to indicate that an element is pending it is prefixed with `!`, and to indicate that the element is excluded it is prefixed with `%`. For instance, a computational data-element `a` with label `A`, having `true` as expression E would have the following syntax:

- `(a:A)[true]`, if included and not pending;
- `%(a:A)[true]`, if excluded and not pending;
- `!(a:A)[true]`, if included and pending; and finally,
- `%!(a:A)[true]`, if excluded and pending.

Getting back to our example of the veterinarian clinic, consider the following example of a data-element with local identifier `createAppointment`: it is written in the abbreviated syntax (it has no markings field) and has no prefix, being possible to conclude that this data-element is both included and not pending in the process; it is also an input data-element (it has a type T field, explicit by the use of `[?: ... ]`), and has record `{clientID:String, vetID:String}` as input type.

```
(createAppointment:DataIN) [?:{clientID:String, vetID:String}]
```

Besides data-elements, ReDa processes are rich in relationships shared between them. There are explicit relationships present in the process definition – control relationships R and data relationships Y – and implicit relationships between data-elements, created by data dependencies or nested identifiers, for instance. A control relationship R can be one

of six relationships, connecting sets of data-elements ϕ and ϕ' and having a guard E as a conditional boolean expression (the syntax of control relationships can also be abbreviated by not having a guard if its (default) value is `true`). Consider the following examples of ReDa control relationships that make use of our veterinarian clinic example theme:

1. condition $\phi \text{-}[E]\rightarrow\bullet\ \phi'$ (`-->*`), indicating that if `E` evaluates to `true` data-elements on the right cannot execute unless each data-element on the left is either marked not included or executed. Consider for instance the condition between `sedatePet` and `operatePet`:

   ```
   sedatePet -[pet.asleep]->* operatePet
   ```

   In this example, the `operatePet` data-element can only be executed either if there is no `sedatePet` event, or once it is previously executed and if the guard `pet.asleep` evaluates to `true`, meaning that to be able to perform an operation it is either not necessary to sedate him, or it is required to do so and wait until he is asleep. Note that there is a data dependency present between the data-element `pet` and this relationship (it is a free name in the expression E of the relationship guard), as whenever `pet` is executed it is necessary to re-evaluate all the relationships that are dependent on its `asleep` property, to know if the guard evaluates to `true`.

2. milestone $\phi \text{-}[E]\rightarrow\diamond\ \phi'$ (`--<>`), indicating that if `E` evaluates to `true` data-elements on the right cannot execute unless each data-element on the left is either marked not included or not pending. Consider the following example where it is only possible to execute `operatePet` if the data-element `appointmentDescription` is not pending or not included:

   ```
   appointmentDescription --><> operatePet
   ```

   Note that there may exist an `appointmentDescription` data-element included in the process and to perform an operation without executing it, but if this description has its `pending` property set to `true` (as it may contain valuable information that would otherwise jeopardize the operation, for instance, some medicine allergies) it is mandatory to execute it first. As there is no guard E it is possible to conclude that its value is `true` by default, and therefore omitted.

3. response $\phi\bullet\text{-}[E]\rightarrow\phi'$ (`*-->`), indicating that if `E` evaluates to `true` whenever some data-element on the left executes, all data-elements on the right become marked pending;

   ```
   callOwner *-[appointment.daysPassed > 3]-> callPETA
   ```

   In this example, the guard is evaluated to `true` if and only if the property `daysPassed` of the `appointment` data-element is greater than three, and whenever `callOwner`

is executed the `callPETA` data-element is set to pending, meaning that if an owner does not check-out its pets from the appointment in three days after it is finished, a call is eventually made to an organization that protects animals. There is once more a data dependency between the data-element `appointment` and this control relationship, as whenever `appointment` is executed the boolean value of the guard condition `appointment.daysPassed > 3` may change.

4. exclusion $\phi \dashv[E]\mapsto\!\!\!\!\%\ \phi'$ (`-->%`), indicating that whenever some data-element on the left executes and if `E` evaluates to `true`, all data-elements on the right become marked excluded;

```
payAppointment -[ MATCH (owner)-[r:made_appointment]->(appointment)
              WHERE owner.age > 60 WITH count(r) as n WHERE n>5 ]->% stateIncomeTax
```

   This is an example of a dynamic condition to represent that customers above the age of sixty that have made more than five appointments in this veterinarian clinic do not need to pay the state income tax, as whenever the `payAppointment` event is executed, the program checks if the owner has a data relationship of type `:made_appointment` with another data-element, and if the result of the aggregation function `count()` applied on this explicit relationship returns more than five the event `stateIncomeTax` becomes excluded from the process;

5. inclusion $\phi \dashv[E]\mapsto\!\!\!\!+\ \phi'$ (`-->+`), indicating that if `E` evaluates to `true` whenever some data-element on the left executes, all data-elements on the right become marked included;

```
checkOut -->+ MATCH (e) WHERE e?animalShelterSupport RETURN e
```

   This relationship is the semantical opposite of the exclusion relationship, and states that whenever the `checkOut` data-element is executed, all data-elements `e` with type `animalShelterSupport` are going to be included into the process (note that the relationship guard E is `true` by default). This means that whenever a customer is checking-out their pets, it is given them the possibility to contribute to animal shelter support organizations (note also that these data-elements are just included and not pending, and as such their execution is not mandatory); and finally,

6. spawn $\phi \dashv[E]\mapsto\!\!\!\!\gg P$ (`-->>`), indicating that whenever some data-element on the left executes and if `E` evaluates to `true`, a new sub-process is merged into the current process. A special identifier `@trigger` denotes the left-hand side (spawner) element in the context of the new sub-process elements.

```
createAppointment -->> { ReDa sub-process }
```

For instance, in this example whenever `createAppointment` is executed (the control relationship guard E is `true` by default) a new sub-process with new ReDa data-elements D, control relationships R, and data relationships Y is going to be added to the process. In this sub-process context, the key-word `@trigger` refers to the spawner data-element `createAppointment`. Consider its following definition:

```
(createAppointment:DataIN) [?: {clientID:String, vetID:String} ]
```

The `createAppointment` event is an input data-element that is both included and not pending (note that it is being used abbreviated syntax) with type record `{clientID:String, vetID:String}`. Consider now that somewhere in the process the following spawn control relationship exists:

```
createAppointment -->> {
  (appointment:DataOUT)[{clientID:@trigger.value.clientID,vetID:@trigger.value.vetID}],
  (...)
  ;
  (...)
  ;
  (...)
}
```

Whenever `createAppointment` is executed, a new output data-element `appointment` with `{clientID:@trigger.value.clientID,vetID:@trigger.value.vetID}` as expression E is added into the process, whose properties `clientID` and `vetID` are evaluated to the value given to the properties `clientID` and `vetID` of the spawner data-element (`createAppointment`), as the key-word `@trigger` references it.

In other languages, like ReDa's precursor ReSeDa [3], the mechanism for relating data items include nesting and using values as keys to identify other elements in other locations of the state. The graph-based nature of ReDa includes direct data relationships between data-elements that guarantee the integrity of all existing relationships. Data relationships Y of the form $\phi \dashv [n : \ell] \mapsto \phi$ create a link between all combinations of nodes resulting from the queries on the left and right-hand-side. Match expressions (ϕ) closely resemble the notation of *cypher* [9] for graph queries, by identifying nodes and relationships via patterns (Q, the `match` clause) and filtering and aggregating the results (ψ, the restriction after the `where` clause that is applied on Q). For instance, it is possible to say that the data-element with the local identifier `john` has a data relationship of type `:is_owner` with all the data-elements of type `pet` where their property `owner` is "john":

```
john -[:is_owner]-> MATCH (p) WHERE p?pet AND p.owner="john" RETURN p
```

Finally, the expression language underlying ReDa includes constructor and destructor expressions for all the data-types referred above, predefined functions, and include the use of match expressions to enable the runtime manipulation of data-elements and their attributes.

3.2 Semantics

The formal semantics of ReDa is defined as a transition system, where states are the data-elements and their respective relationships combined with their marking information, and the transitions are events corresponding to executions (i.e. value updates) of input data-elements. To define this transition system, two functions are required:

- One that determines, for given a ReDa program state, which data-elements are currently executable (enabled), as explained in 3.2.1;
- One that determines, for given a ReDa program state and a data-element, what is the next state after executing said data-element (transitions), as explained in 3.2.2.

3.2.1 Enabledness

For a data-element ρ to be `enabled` it must satisfy that:

- Every data-element ρ' that is a condition for ρ must be either excluded or previously executed;
- Every data-element ρ' that is a milestone for ρ must be either excluded or not pending;
- Every boolean expression `E` present on the guard of a condition or milestone relationship for ρ must be evaluated to `true`, and finally,
- The data-element ρ must itself be included in the process.

The $\mathsf{enabled}_P(\rho)$ function makes this check. For the data-element ρ in the process P its control relationships are traversed, looking for conditions and data-elements that might prevent it from being enabled, and its semantics are defined by the following two cases:

$$
\begin{aligned}
&\mathsf{enabled}_P(\rho) \triangleq \\
&\quad \forall \phi \dashv [E] \rightarrowtail\!\!\bullet\ \phi' \in \overline{R}.\rho \in [\![\phi']\!]_P \implies ([\![\phi[\mathsf{included}]]\!]_P = [\![\phi[\mathsf{included} \wedge \mathsf{executed}]]\!]_P) \wedge (\!|E|\!)_P = \mathsf{true} \\
&\quad \wedge \\
&\quad \forall \phi \dashv [E] \rightarrowtail\!\!\diamond\ \phi' \in \overline{R}.\rho \in [\![\phi']\!]_P \implies ([\![\phi[\mathsf{included}]]\!]_P = [\![\phi[\mathsf{included} \wedge \neg\mathsf{pending}]]\!]_P) \wedge (\!|E|\!)_P = \mathsf{true}
\end{aligned}
$$

This first case states that for each condition relationship existing on the set of all control relationships R of the program `P`, if ρ belongs to the set of data-elements ϕ', then its included property must evaluate to the same boolean value as the logical evaluation of its included and executed properties, which must be `true` for the logical operation to be

`true` as well. Additionally, for the data-element ρ to be considered, its conditional guard `E` needs to evaluate to `true`, and therefore the result of $\texttt{true} \wedge \texttt{true}$ is `true`. The same logic is applied to the milestone relationship, changing only the fact that where it is read "executed" it now reads "not pending". The element ρ is considered `enabled` only if both these cases are evaluated to `true`.

3.2.2 Transitions

In a transition system of a REDA process P, states can be seen as sub-processes P' and transitions to be the update of either a single data-element ρ or multiple data-elements ρ (satisfying a given `match` clause), with respective values v. The set of effects of executing a data-element (or a set of data-elements) ρ on a process P' in the context of a global process P (the set of `effects`P(P', ρ)) are presented next.

The effects are inductively computed based on said data-element ρ control relationships, with each of these base cases treated separately by registering which data-element may cause what particular effect (pending, included, or excluded), and to what other data-elements it is applied.

If the data-element to be executed (ρ) belongs to a set of data-elements ϕ (the left side) of a response relationship, inclusion relationship or an exclusion relationship, all data-elements (ρ') belonging to the set of data-elements ϕ' (the right side) are going to have their pending property set to `true`, their included property set to `true`, or their included property set to `false`, respectively. If the control relationship is a spawn relationship, then whenever a data-element ρ belonging to the set of data-elements ϕ is executed, a new sub-process P' is created, where the name identifier `@trigger` refers to the spawner event(s) ρ in this new context.

$$\mathsf{effects}_P(\overline{R}) \triangleq \cup_{R \in \overline{R}} \mathsf{effects}_{P'}(R)$$

$$\begin{aligned}
\mathsf{effects}_{P'}(\phi \bullet\!\!-[E]\!\!\rightarrow \phi') &\triangleq \{(\rho, (\mathsf{pend}, \rho')) \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P\} \\
\mathsf{effects}_{P'}(\phi -[E]\!\!\rightarrow\!\!+\ \phi') &\triangleq \{(\rho, (\mathsf{incl}, \rho')) \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P\} \\
\mathsf{effects}_{P'}(\phi -[E]\!\!\rightarrow\!\!\%\ \phi') &\triangleq \{(\rho, (\mathsf{excl}, \rho')) \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P\} \\
\mathsf{effects}_{P'}(\phi -[E]\!\!\twoheadrightarrow P') &\triangleq \{(\rho, (\mathsf{spawn}(P'\{^{\rho}/_{\mathsf{trigger}}\}))) \mid \rho \in [\![\phi]\!]_P\}
\end{aligned}$$

When said effects δ are applied to the data-element(s) ρ of the program P, they have their markings field modified accordingly to the `effects` function on program P', as presented next.

$$P \lhd (\delta, \rho) \triangleq P'$$

$$\begin{array}{rcl}
(\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{pend}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,i,\mathtt{t},v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[E]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{pend}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[E]{:}(h,i,\mathtt{t},v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{incl}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,\mathtt{t},p,v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[E]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{incl}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[E]{:}(h,\mathtt{t},p,v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{excl}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h,\mathtt{f},p,v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[E]{:}(h,i,p,v)); \overline{R}; \overline{Y} \lhd (\mathsf{excl}, \rho) & \triangleq & (\overline{D}, (n_\rho : \ell)[E]{:}(h,\mathtt{f},p,v)); \overline{R}; \overline{Y} \\
\overline{D}; \overline{R}; \overline{Y} \lhd (\mathsf{spawn}(\overline{D'}; \overline{R'}; \overline{Y'}), \rho) & \triangleq & \overline{D}, \overline{D'\sigma}; \overline{R}, \overline{R'\sigma}; \overline{Y}, \overline{Y'\sigma} \\
 & & \sigma \text{ is fresh and closes } \overline{D}
\end{array}$$

The first two cases (the same effect is studied when applied on an input data-element and on an output data-element) convey that when the pending effect is applied on a data-element ρ its pending property is set to `true`. The next two pairs of cases are the opposite from each other: whereas the inclusion effect applied on ρ forces the boolean value of the included property to `true`, the exclusion effect forces it to `false`. In the last case, when the effect of a spawn relationship is applied by ρ, the substitution σ replaces the names of events defined by $\overline{D'}$ with fresh event identifiers, in order to instantiate the sub-process. The same names are free in $\overline{R'}$ and $\overline{Y'}$.

Finally, the transition rules presented next are used to define the transition system, making a distinction between computation events (rule `compute`) and input events (rule `update`). The former transitions require which data-element is going to be executed (ρ), whereas the latter requires both the data-element ρ and an input value v (labeled $\rho(v)$). In both cases the selected data-element ρ must be enabled, and the effects of the data-elements execution are applied to a process P' where the target data-element is already (re)computed/updated. Computing a transition involves in either case three steps:

- The data-element in question (ρ) must be enabled;
- Then, for an input data-element ρ, the value of ρ is updated with the value of input v ($\mathtt{update}_\rho^P$ (P, ρ, v)). For a computation data-element, the computation is executed ($\mathtt{compute}_\rho^P$ (P, ρ));
- Finally, the effects of executing ρ are computed and applied ($\mathtt{effects}_P$' (ρ)).

$$\frac{\mathsf{enabled}_P(\rho) \quad P' = \mathsf{update}_P(\rho, v) \quad \delta = \mathsf{effects}_{P'}(\rho)}{P \xrightarrow{\rho(v)} P' \lhd \delta}$$

$$\frac{\mathsf{enabled}_P(\rho) \quad P' = \mathsf{compute}_P(\rho) \quad \delta = \mathsf{effects}_{P'}(\rho)}{P \xrightarrow{\rho} P' \lhd \delta}$$

3.2.3 Data expressions

The semantics of an expression E on a program P, written $(\!|E|\!)_P$, are defined by the following cases:

$$
\begin{aligned}
(\!|c|\!)_P &\triangleq c \\
(\!|\mathbf{1}|\!)_P &\triangleq \mathbf{1} \\
(\!|\bot|\!)_P &\triangleq \bot \\
(\!|\phi|\!)_P &\triangleq [\![\phi]\!]_P \\
(\!|\psi|\!)_P &\triangleq \mathsf{true} && \text{if } [\![\psi]\!]_P \neq \emptyset \\
(\!|\psi|\!)_P &\triangleq \mathsf{false} && \text{if } [\![\psi]\!]_P = \emptyset \\
(\!|E{:}attr|\!)_P &\triangleq v && \text{if } (\!|E|\!)_P = \{\ldots, attr : v\} \\
(\!|f(E_1,...,E_n)|\!)_P &\triangleq f((\!|E_1|\!)_P,...,(\!|E_n|\!)_P)
\end{aligned}
$$

The first three cases are trivial, as the semantic evaluation of a constant, the unit value, or the undefined value is the constant, unit value, and undefined value, respectively. The last two cases state that the semantic evaluation of an attribute is its current value, and the semantics of a function with expressions is the function itself applied to the individual semantic evaluation of its expressions.

The semantics of node queries, written $[\![\phi]\!]_P$, is defined by analyzing the possible cases for ϕ and relying on the semantics of match expressions, written $]\!|\psi|\![_P$, where σ is a substitution mapping names to values, $\sigma(x)$ is the value assigned to x in substitution σ, and $E\sigma$ is the application of the substitution in expression E.

$$
\begin{aligned}
[\![\psi \ \mathsf{Return}\ x]\!]_P &\triangleq \{\ \sigma(x)\ \mid\ \sigma \in]\!|\psi|\![_P\} \\
]\!|\mathsf{Match}\ Q\ \mathsf{Where}\ E_1|\![_P &\triangleq \{\ \sigma\ \mid\ \sigma \in \|Q\|_P \wedge (\!|E_1\sigma|\!)_P = \mathsf{true}\} \\
]\!|\psi\ \mathsf{With}\ pipe\ \mathsf{Where}\ E_2|\![_P &\triangleq \{\ \sigma'\ \mid\ \sigma' \in]\!|\psi|\![,\ (agg\ \mathsf{AS}\ n) \in pipe,\ v = \lceil agg \rceil^{\sigma'}\ \wedge\ (\!|E_2\sigma'\{^v/_n\}|\!)_P = \mathsf{true}\}
\end{aligned}
$$

The first case states that the semantic evaluation of the node query is the assignment of all substitutions σ possible to the variable x. The semantics of the second case are the substitutions returned by the semantic evaluation of the pattern Q (Section 3.2.3.1) that are `true` on the expression *E1*, and lastly, the semantics of the last case are all the substitutions σ' belonging to ψ (the former case), where `n` are new names for the aggregations belonging to the pipe, `v` are the results of the aggregation functions given all substitutions σ', and where the evaluation of the substitutions obtained by replacing `n` with all values `v` on σ' to *E2* are `true`.

3.2.3.1 Patterns

The semantics of pattern evaluation $\|Q\|_P$ are defined by the three cases below [1]:

$$\begin{aligned}
\|n\|_P &\triangleq \{\ n \mapsto \rho \mid P = \overline{D};\overline{R};\overline{Y},\ n_\rho \in \overline{D}\} \\
\|n \dashv[\cdot]\vdash n'\|_P &\triangleq \{\ \{n \mapsto \rho, n' \mapsto \rho'\} \mid P = \overline{D};\overline{R};\overline{Y} \wedge (n_\rho \dashv[\cdot]\vdash n'_{\rho'}) \in \overline{Y}\} \\
\|Q_1, Q_2\|_P &\triangleq \{\ \sigma_1\sigma_2 \mid \sigma_1 \in \|Q_1\|_P,\ \sigma_2 \in \|Q_2\|_P,\ \sigma_1 /\!/ \sigma_2\}
\end{aligned}$$

The first case means that the semantic evaluation of a variable n without relationships returns all data-elements of the program. Next, the second case returns all possible permutations where the relationship between n and n' belongs to the set of all data relationships in the program, and lastly, the union of two patterns returns the union of the substitution in each pattern where all common names have the same value.

3.2.3.2 Aggregation Functions

The semantics of the aggregation functions, written $\lceil agg \rceil^\sigma$ where σ represents the possible substitutions to E, are defined below:

$$\begin{aligned}
\lceil n \rceil^\sigma &\triangleq \sigma(n) \\
\lceil count(E) \rceil^\sigma &\triangleq \#\{\ \sigma \mid \sigma \in [\![\psi]\!]_P \wedge (\!|E\sigma|\!)_P \neq \mathsf{null}\} \\
\lceil max(E) \rceil^\sigma &\triangleq max(\{\ (\!|E\sigma : attr|\!)_P \mid \sigma \in [\![\psi]\!]_P\}) \\
\lceil min(E) \rceil^\sigma &\triangleq min(\{\ (\!|E\sigma : attr|\!)_P \mid \sigma \in [\![\psi]\!]_P\}) \\
\lceil sum(E) \rceil^\sigma &\triangleq sum(\{\ (\!|E\sigma : attr|\!)_P \mid \sigma \in [\![\psi]\!]_P\}) \\
\lceil avg(E) \rceil^\sigma &\triangleq avg(\{\ (\!|E\sigma : attr|\!)_P \mid \sigma \in [\![\psi]\!]_P\})
\end{aligned}$$

The first case returns all the possible substitutions (σ) for the variable n. Next, `count()` returns the number of possible substitutions σ on all free names of E different from `null`, and `max()` returns the maximum value out of all values returned by doing each possible substitution σ on all free names of E. The same logic is applied on `min()`, `avg()` and `sum()`, but returning the lowest value, the average of all values and the sum of all values, respectively.

[1] $\sigma\sigma'$ denotes the union of the two substitutions. $\sigma /\!/ \sigma'$ means that all common names have the same value, and thus the union is defined.

3.3 ReDa by example

In this section, ReDa is exemplified with the help of a running example of a library able to create records of authors and their books in a database, and also manages loans to users using their names (this example was chosen rather than the veterinarian clinic as it provides a richer approach to ReDa's syntax and semantics, but its modeling and respective translation to *cypher* is present at Annex II). For the sake of briefness, users are identified only by their name, and no further information and control is given. ReDa constructions and operations are gradually re-visited.

A ReDa process definition simultaneously introduces the data model and the control flow of a software system. The language that defines the process is the actual programming language used to define a system's behavior. The data model comprises interdependent nodes, interconnected in a graph denoting either input or computation data-elements. Input data-elements define typed points of entry for data that are linked to external systems (e.g. web services) or user interfaces capable of any form of interaction and data input. Output data-elements denote a value computed from their enclosed expression, and can reactively refer to other (computation or input) elements. Thus, each output data-element is up-to-date with its references like a spreadsheet cell.

The view of the data model provided by a ReDa process is completed by the definition of explicit data relationships that can be queried together with the data-elements and used explicitly in control flow definitions. Furthermore, syntactic dependencies between data-elements, from using their identifiers and properties, define an implicit control flow layer that is complemented by the definition of explicit control flow relationships between nodes. Both data and control relationships can be dynamically controlled by boolean conditions (guards) on nodes and their properties.

Control relationships in ReDa are inherited from DCR graphs and their process definition language. They denote dependencies between events (condition, response, or milestone relationships), and the control over the visibility (enabledness) of data-elements (including or excluding them from active participation in the process). Lastly, ReDa supports the introduction of sub-processes, allowing the creation of new nodes and relationships as an essential mechanism for the dynamic creation of new control and data structures. In the end, data-elements are event-like structures in the sense of DCR graphs. The occurrence of events in ReDa is always associated with the input of new data or with the (re)computation of values.

Consider the following ReDa fragment defining a ReDa process with three main sections: node declarations, control relationship declarations, and data relationship declarations [2].

[2] Notice that in the examples a semi-colon separates sections and a comma separates individual definitions.

```
1 (createAuthor:DataIN) [?: { authorName:String }]
2 ;
3 createAuthor -->> {
4   (author:DataOUT)[{ name:@trigger.value.authorName }]
5   ;
6   ;
7 }
8 ;
```

Line 1 declares an input data-element and declares the identifier `createAuthor`, whose scope are the definitions below, as accepting values as input of type record `{authorName:String}`. The second declaration (line 3) denotes the spawning of a new ReDa sub-process, containing an output data-element with the identifier `author`, visible inside the process on the right-hand side of a spawn relationship (`-->>`). The enclosed expression of type record denotes the value associated with this computation node, where `@trigger` is evaluated once in a call-by-value strategy (by copying the value and not a reference to the original location). The third and empty section in the snippet above is reserved for the declaration of explicit data relationships. Using this process and executing the `createAuthor` input data-element – with values `"Tolkien"` and `"Philip K. Dick"` – the process now contains the data-elements depicted in Figure 3.2.

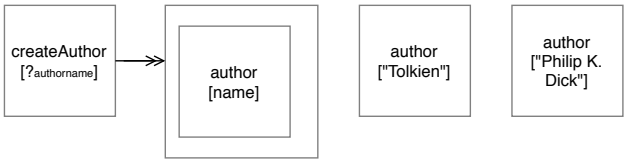

Figure 3.2: Process state after the creation of two authors.

Each data-element in the graph is stateful and has an associated marking, composed of three properties: executed, included, and pending. Like in DCR graphs, these properties specify if and, in this case, how many times a given data-element has already been executed, whether it is currently included in the process activity and if it is a pending event, required to happen at a given time, respectively. When possible, these markings are syntactically replaced by symbols to differ from the default ones (included and not pending), by using a `%` prefixed to the definition of a data-element representing its exclusion in the graph, and a `!` indicating that the data-element has a pending computation. For instance, the expanded syntax of the declaration of the input data-element `createAuthor` is the following:

```
(createAuthor:DataIN) [?:{authorName:String}] : {executed:0,included:true,pending:false}
```

To continue the example, the sub-process will be extended with more declarations to allow the creation of books in the process and underlying data model.

```
(createAuthor:DataIN) [?: { authorName:String }]
;
createAuthor -->> {
  (author:DataOUT) [{ name:@trigger.value.authorName }],
  (createBook:DataIN) [?:{ bookTitle:String, isbn:String }]
  ;
  createBook -->>{
    (book:DataOUT) [{bookTitle:@trigger.value.bookTitle, isbn:@trigger.value.isbn, author:
        author.value.name }]
    ;

    ;
    author -[:WROTE]-> book
  }
  ;
}
;
```

This snippet states that, in case of an occurrence of a `createAuthor` data-element, it also defines a new instance of an input data-element `createBook`, each associated to a different `author` data-element, as well as a new spawn relationship in that nested sub-process, activated whenever `createBook` is executed for that particular value of `author`. Also, a new data relationship labeled `WROTE` is created between the `author` and the `book` nodes at the end, specifying an explicit relationship between them and meaning that this new book was written by that specific author created in its upper scope.

Notice that, via a nested reference inside the spawn of `createBook`, there is a relationship between every `createBook` data-element and the `author` data-element defined in the same context. This syntactic relationship is preserved at runtime very much like a closure preserves the values of the free names of the function originating it. So, each author in the process has a related action or entry point in the system to create its books. Using this process and executing once again the `createAuthor` input data-element – with values `"Tolkien"` and `"Philip K. Dick"` – but now also adding a book for each one of them, the process contains the data-elements depicted in Figure 3.3.

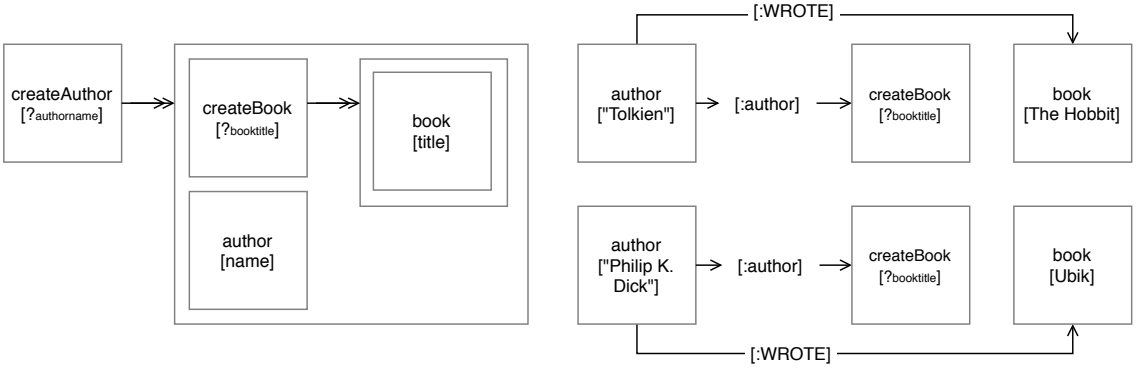

Figure 3.3: Process state after the creation of two authors and two books.

```
1  (createAuthor:DataIN) [?:{authorName:String}]
2  ;
3  createAuthor -->> {
4    (author:DataOUT) [{name:@trigger.value.authorName}],
5    (createBook:DataIN) [?:{bookTitle:String, isbn:String}]
6    ;
7    createBook -->>{
8      (book:DataOUT) [{bookTitle:@trigger.value.bookTitle,
9                  isbn:@trigger.value.isbn, author:author.value.name}],
10     (loanBook:DataIN) [?:{username:String}]
11     ;
12     loanBook -->% loanBook,
13     loanBook -->>{
14       (loan:DataOUT) [{user:@trigger.value.username}],
15       !(returnBook:DataIN) [?]
16       ;
17       returnBook -->% returnBook,
18       returnBook -->+ loanBook
19       ;
20       loan -[:BOOK]-> book,
21       returnBook -[:LOAN]-> loan
22       }
23     ;
24     author -[:WROTE]-> book
25     }
26     ;
27 }
28 ;
```

Next, an input data-element named `loanBook` is introduced into the sub-process of each book. To simplify, this input data-element only requires the name of the user requesting to loan the book. There are also two new control relationships: the first one states that whenever `loanBook` executes it excludes itself from the process (line 12), and the second is a spawn relationship (line 13) that adds a `loan` data-element whose value is the name given as input to `loanBook` node, as well as pending input data-element `returnBook` that needs to be fired when the user wants to terminate a loan in the library.

Two control relationships are also introduced by this sub-process: the first says that `returnBook` excludes itself (line 17), i.e. it cannot happen twice, and another that includes `loanBook` back (line 18), allowing for new loans to happen.

The diagram that corresponds to this last complete version of the ReDa library process is shown in Figure 3.4. It is possible to see that this includes the classic DCR graph relationships, plus graph relationships between data-elements. One detail that is not visible in this diagram is the data dependencies introduced by expressions in computation elements (node instances for authors, books, and loans are not shown here). ReDa, as presented above, is a core calculus that is at the heart of a process-aware system.

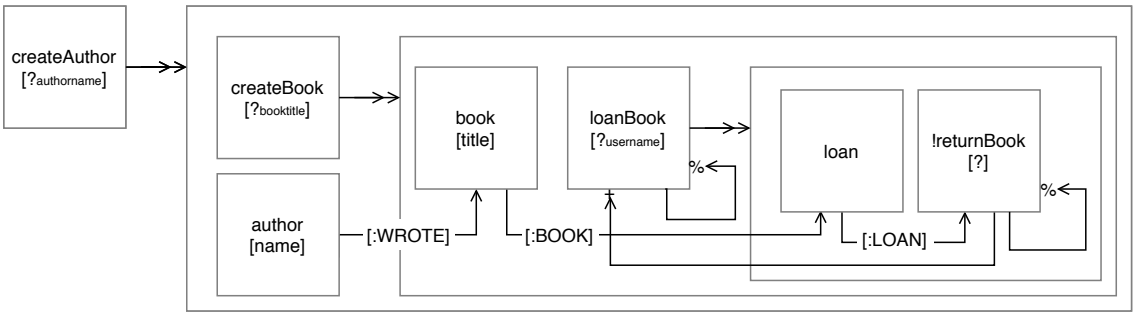

Figure 3.4: Graphical representation of the example.

CHAPTER 4

Compilation Procedure

Whilst ReDa (Chapter 3) is a declarative approach to model business processes, where its semantics and data are unified, applicability is obtained in the form of business process modeling systems and their engines (Section 2.2). In this chapter we present the procedure to systematically transform ReDa processes into *cypher* scripts [9] (the native language of the *neo4j* graph-database), maintaining all process behavior and semantics in the form of database triggers. This procedure represents the first step to achieve process automation.

The approach is based in a design where executions (performing a given task in a process) are translated to update queries in the database, and where data-elements and their properties can be queried by external systems. The execution of data-elements is conveniently guarded by the database against inadmissible operations, like the execution of an element that is not enabled. The remaining process behavior is encoded to *cypher* and executed by the database engine. The compilation procedure that systematically transforms a ReDa process into *cypher* code, denoting its behavior, is going to be presented next.

The structure of a *cypher* script is a flat list of node and relationship declarations, graph queries, update commands, and other (trigger) definitions. A hoisting mechanism that statically defines explicit relationships between node instances in the graph of the target system to represent nested relationships between node identifiers is used to transform the nested structure of ReDa processes into the flat structure of *cypher* code. The result is a *cypher* script with top-level creation commands for top-level nodes and data relationships, and triggers that represent the delayed inner definition contexts of spawn relationships.

A ReDa process is translated into a four-part *cypher* script containing: `(i)` a set of queries that are used to fetch related nodes from other contexts to be used in local (or inner) definitions – empty at the top level; `(ii)` a set of node definitions that correspond to local definitions of input and computation nodes; `(iii)` a set of relationship definitions

that correspond to data dependencies, control relationships, and data relationships defined in the current process; and finally, `(iv)` in the case of triggers associated to a data-element, one update command that (re)evaluates the node's expression with relation to the nodes it depends on. There is one special kind of trigger, which is depicted on line six of Figure 4.1, that contains the constraints and verifications necessary for the semantics of ReDa to be applied, and to allow matches and dynamic conditions. Nested relationships come from the definition of sub-processes (introduced by spawn relationships), which are static and fixed in a ReDa process. Thus, all name dependencies are statically resolved and generate *cypher* relationships and queries that correctly implement a static binding strategy.

```
1  -- Top level --
2  ii) creation of nodes
3  iii) creation of relationships between nodes
4
5  -- Triggers --
6  Main_trigger { (...) }
7
8  Trigger_1 {
9    i) node lookup
10   ii) creation of nodes
11   iii) creation of relationships between nodes
12   iv) updates
13 }
14 (...)
```

Figure 4.1: General structure of a *cypher* script.

Sub-processes are compiled into *cypher* code enclosed in triggers, which are associated, by the compiler, to changes (execution) in the spawner nodes. Such triggers are executed whenever the corresponding data-element is requested to execute. The nesting of processes results in the nesting of name declarations and the use of names defined in outer contexts. The hoisting mechanism for processes includes the encoding of non-local names in persistent relationships between nodes, created upon the execution of the outer context, and the retrieval of the relationships upon the execution of the trigger that represents the inner context. This linked structure is defined between node instances during the execution of the process, depicting the dynamic nature of the relationships.

Finally, ReDa control relationships are translated to relationships between node instances. The enabledness check is translated into explicit validations that check if any preceding element (using a relationship with labels `:condition` or `:milestone`) is included and not executed in the case of condition relationships, or included and pending in the case of milestone relationships. The effects of execution via the response, includes and excludes relationships are translated to *cypher* queries that search for this kind of relationships between node instances and modifies the marking of the target node accordingly. The enabledness check and the subsequent execution of effects is performed by a main trigger which may abort any transaction in case of error.

```
1  (a:A)[?:Number],
2  (b:B)[a.value+1]
3  ;
4  b -->% a,
5  a -->> {
6    (c:C)[ {x:a.value+b.value, y:@trigger.value} ]
7    ;
8    c -->> {
9      (d:D)[c.value.x+c.value.y+a.value]
10     ;
11     ;
12   }
13   ;
14 }
15 ;
```

Figure 4.2: Toy example to illustrate the compilation of ReDa to *cypher*.

To illustrate the compilation procedure, consider the ReDa example of Figure 4.2, gathering the main cases of the compilation function (data dependencies, nested processes, and subprocesses) that we next translate into a *cypher* script. The resultant script of this example is simplified for now, not taking into account matches or dynamic conditions, which are approached in Section 4.1. First, there are two top-level node (data-element) definitions that correspond to the defined name, label, and default values of the markings.

```
1  CREATE (a_0:A{reda_id="a_0", executed:0, included:true, pending:false, value:0})
2  CREATE (b_1:B{reda_id="b_0", executed:0, included:true, pending:false})
```

Line 1 is the declaration of the input node `a`, which includes the initialization of the value to the default value appropriate for the data-type (`0`). Node `b` is declared in line 2 and its value is left uninitialized since it depends on node `a` and cannot be evaluated at this stage. Notice the alpha-renaming of node names with fresh identifiers (`a_0`, `b_1`) to avoid clashing between different declaration contexts.

Consider the syntactic dependencies between `a` and `b` created by the expression in the definition of `b`, and because `a` spawns a subprocess using `b`. The compiled code of the subprocess is emitted in a delayed *cypher* context in the trigger shown below. Explicit node relationships that define a name substitution are used in order to maintain the static resolution of names:

```
3  CREATE (a_0)-[:a]->(a_0)
4  CREATE (a_0)-[:a]->(b_1)
5  CREATE (b_1)-[:b]->(a_0)
```

Line 4 means: node `a_0` in this context is the correct substitution for the free name `a` in all sub-processes (and expressions) of node `b_1`. The data dependency of `b` on `a` is then

reified into a control relationship dependency as follows: `b` cannot be executed without first having executed – and thus gotten a value for – `a` (line 6); and equally clearly, whenever the value of `a` changes, `b` must be re-computed to reflect that change in its value (line 7). That is, the following condition and response are added:

```
6  CREATE (a_0)-[:condition]->(b_1)
7  CREATE (a_0)-[:response]->(b_1)
```

These relationships establish the essence of the reactive behaviour of ReDa, following the semantics of ReSeDa (Section 2.1.3.4), and the mechanics are akin to spreadsheet semantics: updating the "cell" `a` forces a re-computation of the value of `b`. The excludes control relationship on line 4 of the ReDa program (Figure 4.2) is compiled almost as is (as seen below).

```
8  CREATE (b_1)-[:excludes]->(a_0)
```

This concludes the top-level declarations of nodes and relationships in the translated program. Next, the remainder *cypher* program comprises a trigger declaration that implements the necessary enabledness verification and the computation of effects (inclusion, exclusion, responses) upon a successful execution of any data-element (the `main_trigger` – note that it is still a simplified approach, not taking into account matches and dynamic relationships), and trigger declarations that comprise the expected process behaviour upon the execution of specific data-elements.

```
9  CALL apoc.trigger.add(‘‘Main Trigger’’,
10  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed")
11     as prop WITH prop.node as n WHERE n.executed>0
12
13     CALL apoc.util.validate(n.included=false, ‘‘EVENT IS NOT INCLUDED’’, [])
14     CALL apoc.util.validate(EXISTS((n)<-[:condition]-({included:true, executed:0})),
15              ‘‘EVENT HAS A CONDITION UNSATISFIED’’, [])
16     CALL apoc.util.validate(EXISTS((n)<-[:milestone]-({included:true, pending:true})),
17              ‘‘EVENT HAS A MILESTONE UNSATISFIED’’, [])
18
19     SET n.pending=false WITH n
20     OPTIONAL MATCH (n)-[:response]->(t) SET t.pending = true WITH n
21     OPTIONAL MATCH (n)-[:excludes]->(t) SET t.included = false WITH n
22     OPTIONAL MATCH (n)-[:includes]->(t) SET t.included = true
23
24 RETURN 1 ’,{phase:‘‘before’’});
```

neo4j triggers are defined using the plugin `apoc` that receives a definition that identifies the node being changed and some basic filters on it (lines 9-11). The trigger is set to fire before the commit occurs (line 24) which allows to abort the modification in case of

error. This trigger first checks for the enabledness property of a node – notice that it must be included, and all predecessors in condition relationships that are included nodes must have executed in the past, and all predecessors in milestone relationships that are included must not be pending (lines 13-17). Since executing a node is an operation external to the system that updates the `executed` field of a data-element, these conditions represent the necessary guards that ensure ReDa semantics (Section 3.2). Whenever a given data-element is executed, its `pending` property should be set to false (line 19), and the effects of the remaining DCR inspired relationships are applied here by updating the nodes that are related to the triggered node (lines 20-22).

We next present the behaviour of executing data-elements, including the spawning of sub-processes, for each of the node definitions in the program. Such triggers are fired whenever the associated `executed` property is changed. This is visible in the definition below including the condition filtering the node's `reda_id`, and also checking if the property's value is strictly positive (as a convention, the value of this property should always grow), so that the trigger is not fired upon the creation of the node (lines 25-27). These triggers contain the compiled code for the actions to be executed when related nodes are (re)evaluated, including the spawning of sub-processes. Consider the example of node `a_0`, compiled from the computation node `a` in the example.

```
25 CALL apoc.trigger.add(‘‘When a_0 happens’’,
26 ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’)
27   as prop WITH prop.node as n WHERE n.reda_id="a_0" AND n.executed>0
28
29   MATCH (a_0)-[:a]->(n)
30   MATCH (b_1)-[:b]->(n)
31
32   CREATE (c_2:C{reda_id="c_2", executed:0, included:true, pending:false, value_y:n.value})
33   CREATE (a_0)-[:a]->(c_2)
34   CREATE (b_1)-[:b]->(c_2)
35   CREATE (c_2)-[:c]->(c_2)
36   CREATE (b_1)-[:condition]->(c_2)
37   CREATE (b_1)-[:response]->(c_2)
38   CREATE (a_0)-[:condition]->(c_2)
39   CREATE (a_0)-[:response]->(c_2)
40
41   RETURN 1 ’,{phase:‘‘before’’});
```

This trigger starts by establishing the correct substitution for the free names of the sub-process of `a_0`, by querying the nodes that represent `a` and `b` in this context (lines 29-30). These relationships match the relationships created at the top-level (lines 4-5 of the translated script) – recall that both `a` and `b` are present as dependencies of `c`'s value in the subprocess created upon executing `a` (Figure 4.3).

```
(...)
a -->> {
  (c:C)[ {x: a.value + b.value, y: @trigger.value} ]
  ;
  c -->> {
    (d:D)[c.value.x + c.value.y + a.value]
    ;
    ;
  }
  ;
}
(...)
```

Figure 4.3: Excerpt of the ReDa process presented in Figure 4.2.

Line 32 includes the local node definition for identifier `c`, here alpha-renamed to `c_2`, and includes the partial evaluation of the node expression `{x:a.value+b.value, y:@trigger.value}`. There are two observations at this stage to be made: the first is the flattening of record values in *cypher* with relation to ReDa, the second is that field `x` depends on other nodes and cannot be computed at this stage, while field `y` depends on the trigger node and should only be evaluated using a call-by-value strategy by copying the values to the current node, meaning `value_y` property receives the value of the trigger node `n.value`. The call-by-need semantics of node dependencies (such as the `value_x`) is depicted in the trigger handling changes in node `c_2`. Lines 33-39 establish data dependencies between identifiers for the inner scope of expressions and subprocesses as described at the top level (notice that the identifier `a` crosses more than one syntactic context level and direct links are created by all stages).

The trigger that handles node `b` is quite simpler since it is not used to spawn subprocesses. The value of node `b` depends on the value of node `a`, so its trigger basically just (re)computes the value of `b` whenever `a` is executed. Line 46 retrieves the substitution for identifier `a_0` in this context and updates the value attribute of node `b_1` (line 48), guaranteeing that `b`'s value is always equal to `a`'s plus one (Figure 4.4).

```
42 CALL apoc.trigger.add(‘‘When b_1 happens’’,
43 ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’)
44   as prop WITH prop.node as n WHERE n.reda_id="b_1" AND n.executed>0
45
46   MATCH (a_0)-[:a]->(n)
47
48   SET n.value=a_0.value+1
49
50 RETURN 1’,{phase:’before’});
```

```
(a:A)[?:Number],
(b:B)[a.value + 1]
;
(...)
;
```

Figure 4.4: Excerpt of the ReDa process presented in Figure 4.2.

The trigger for the node `c` starts by retrieving the substitutions for names `a`, `b`, and `c`, from the previously created scoping relationships (lines 56-58) and computes the value of `c_2` (line 60) – recall that the value of `c` depends on both `a` and `b`, and both `a` and `c` appear as free names in the value of `d` (Figure 4.3) . Notice once more the flattened structure of the update structure, here solely updating the part of the record in `c_2` that depends on other nodes (call-by-need strategy), as the remainder was initialized on creation (line 32) and does not change (call-by-value strategy). Lines 62-68 represent the creation of data dependencies between identifiers for this scope, as seen on the previous triggers.

```
51  CALL apoc.trigger.add(‘‘When c_2 happens’’,
52  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’)
53    as prop WITH prop.node as n
54    WHERE n.reda_id="c_2" AND n.executed>0
55
56    MATCH (a_0)-[:a]->(n)
57    MATCH (b_1)-[:b]->(n)
58    MATCH (c_2)-[:c]->(n)
59
60    SET n.value_x = a_0.value + b_1.value
61
62    CREATE (d_3:D{reda_id:"d_3", executed:0, included:true, pending:false})
63    CREATE (a_0)-[:a]->(d_3)
64    CREATE (a_0)-[:condition]->(d_3)
65    CREATE (a_0)-[:response]->(d_3)
66    CREATE (c_2)-[:c]->(d_3)
67    CREATE (c_2)-[:condition]->(d_3)
68    CREATE (c_2)-[:response]->(d_3)
69
70  RETURN 1’,{phase:‘‘before’’});
```

Finally, the trigger for node `d`, alpha-renamed to `d_3`, handles the update of its value (line 79) upon substitution of its free names `a` and `c` (lines 76-77). Notice the use of the flattened selection expression for both fields of the record value in node `c`.

```
71  CALL apoc.trigger.add(‘‘When d_3 happens’’,
72  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’)
73    as prop WITH prop.node as n
74    WHERE n.reda_id="d_3" AND n.executed>0
75
76    MATCH (a_0)-[:a]->(n)
77    MATCH (c_2)-[:c]->(n)
78
79    SET n.value = c_2.value_x + c_2.value_y + a_0.value
80
81   RETURN 1’,{phase:‘‘before’’});
```

4.1 Dynamic Relations

It is of essence to talk about helper `dummy` nodes to introduce the notion of matches and dynamic conditions on a *cypher* script. These nodes also have markings but they differ from the regular ones by being always included and having their value restricted to being `true` or `false`. They are used in the following cases:

- `(i)` if there is a relationship without guard where one or both sides have a match, then the `dummy` node's value is always `true` by default, and it has explicit relationships with the data-elements that are the correct substitution for the free names present on either side of the relationship, at that context;
- `(ii)` if there is a relationship with guard then the `dummy` node's value is its logical evaluation at that stage, and it has explicit relationships with the data-elements that are the correct substitution for the free names present on the expression.

```
1  (a:X)[?:],
2  (b:X)[3],
3  (c:X)[true],
4  %(e:Y)[true]
5  ;
6  a -[b.value>0]->> {
7    (d:Y)[true]
8    ;
9    (MATCH (x) WHERE x?Y RETURN x) -[c.value]->+ e
10   ;
11   b -[:relation]-> MATCH (var) WHERE var.executed>2 RETURN var
12 }
13 ;
```

Figure 4.5: Toy example with dynamic conditions and matches.

Figure 4.5 depicts a new example containing dynamic conditions and matches as a way to have multiple data-elements present in one side of a relationship. The first set of top-level creations follow the same logic seen in the previous example. Lines 1-4 on the *cypher* script are the declaration of the input node `a`, `b`, `c`, and `e`. Notice again the alpha-renaming of node names with fresh identifiers (`a_0`, `b_1`, `c_2` and `e_3`) to avoid clashes, that `c_3` starts with its included property set to false (see Section 3.1 for this concrete syntax modifiers), as depicted in the *cypher* script on line 4, and that `a_0` is an input data-element of unit type, meaning it is an input data-element where there is no input to be given other than its actual execution, just like clicking a button in a user interface.

```
1 CREATE (a_0:X{reda_id:"a_0", executed:0, included:true, pending:false, value:null})
2 CREATE (b_1:X{reda_id:"b_1", executed:0, included:true, pending:false, value:3})
3 CREATE (c_2:X{reda_id:"c_2", executed:0, included:true, pending:false, value:true})
4 CREATE (e_3:Y{reda_id:"e_3", executed:0, included:false, pending:false, value:true})
```

Line 5 includes the creation of the helper node `dummy_9`. This `dummy` node has the expression `b.value>0` as guard, and a `:response` relationship with `b_1` (line 6), stating that whenever the latter happens this helper node has to be executed again. The value of helper nodes is either `true` or `false`, and in this case, as it depends on the value of `b_1`, it will be evaluated once the latter is executed. As present in line 7, `dummy_9` is the `:spawnCondition` for the input node `a_0`, meaning that only if this node's value is evaluated to `true` (the relationship guard) the spawn relationship is activated. On lines 8-9 are created the relationships that allow the nodes `b_1` and `e_3` of this context to be fetched on the sub-context created when `a_0` happens and the guard evaluates to `true`.

```
5 CREATE (dummy_9:DUMMY{reda_id:"dummy_9", exp:"b.value>0", executed:0, included:true,
      pending:false})
6 CREATE (b_1)-[:response]->(dummy_9)
7 CREATE (dummy_9)-[:spawnCondition]->(a_0)
8 CREATE (b_1)-[:b]->(a_0)
9 CREATE (e_3)-[:e]->(a_0)
```

This concludes the top-level declarations of nodes and relationships on the *cypher* script. Next we present the delayed ReDa process information which is translated to triggers. However, unlike the previous example, the main trigger is approached last as it is where most changes needed to accommodate dynamic conditions and matches are.

```
10 CALL apoc.trigger.add(‘‘When dummy_9 happens’’,
11  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
12   WITH prop.node as n WHERE n.reda_id=‘‘dummy_9’’ AND n.executed>0
13
14   MATCH (b_1)-[:response]->(n)
15   WHERE b_1.reda_id = "b_1"
16
17   SET n.value = b_1.value > 0
18
19   RETURN 1 as X
20 ’,{phase:‘‘before’’});
```

The trigger activated when `dummy_9` executes is just like any other: queries on lines 14-15 are responsible to fetch the correct substitution for the free name `b_1` on its expression, and in line 17 its value is set to the correct boolean value.

```
21 CALL apoc.trigger.add(‘‘When a_0 happens’’,
22  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
23   WITH prop.node as n WHERE n.reda_id="a_0" AND n.executed>0
24
25   MATCH (b_1)-[:b]->(n)
26   MATCH (e_3)-[:e]->(n)
27   MATCH (dummy_9)-[:spawnCondition]->(n) WHERE dummy_9.value = true
28
29   CREATE (d_4:Y{reda_id:"d_4", executed:0, included:true, pending:false, value:true})
30   CREATE (dummy_5:DUMMY{reda_id:"dummy_5", exp:"true", value:true, executed:0, included:
         true, pending:false})
31   CREATE (dummy_8:DUMMY{reda_id:"dummy_8", exp:"c.value", executed:0, included:true,
         pending:false})
32
33   CREATE (b_1)-[:b]->(dummy_5)
34   CREATE (c_2)-[:response]->(dummy_8)
35   CREATE (e_3)-[:e]->(dummy_8)
36
37   RETURN 1 as X
38 ’,{phase:‘‘before’’});
```

The trigger activated when `a_0` executes starts with queries in lines 25-26 fetching the correct substitutions for the free names `b_1` and `e_3` in this context (remember that ReDa processes have a nested structure and the same identifiers can be used in different levels to refer to different data-elements). The instruction in line 27 guarantees that the guard present on the spawn condition in line 6 of Figure 4.5 is satisfied: the helper node `dummy_9` that is the `:spawnCondition` to `a_0`'s execution must have its value set to `true`, or else the trigger is aborted and the remaining operations are not performed. Assuming that the guard condition is satisfied, in line 29 is created the data-element `d_4` (as present on Figure 4.5 in line 7), and introduces two new helper nodes: `dummy_5` that supports the

data relationship present on line 11 of Figure 4.5, as it has a `match` and needs to have an explicit relationship with all free names that occur on it (in this case is only the free name `b` – line 33); and `dummy_8` supports the inclusion relationship in line 9 of Figure 4.5, having its guard as expression, a `:response` relationship with `c_2` (line 34, as the guard depends on its value), and a relationship with `e_3` to perform the correct substitution (line 35).

```
39 CALL apoc.trigger.add(‘‘When dummy_8 happens’’,
40  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
41   WITH prop.node as n WHERE n.reda_id="dummy_8" AND n.executed>0
42
43   MATCH (c_2)-[:response]->(n)
44   WHERE c_2.reda_id="c_2"
45
46   SET n.value = c_2.value
47
48   RETURN 1 as X
49 ’,{phase:‘‘before’’});
```

Both these `dummy` nodes are very simple in their execution: `dummy_5` is defined only to know the correct substitution to the free name `b`, existing no need to have a trigger for it; and `dummy_8` fetches the node to whom has a `:response` relationship with, updating its value accordingly (lines 43-46).

```
50 CALL apoc.util.validate( EXISTS((n)<-[:condition]-({included:true, executed:0})), ‘‘EVENT
       HAS A CONDITION UNSATISFIED’’, [])
51
52 CALL apoc.util.validate( EXISTS((n)<-[:milestone]-({included:true, pending:true})), ‘‘
       EVENT HAS A MILESTONE UNSATISFIED’’, [])
53
54 CALL apoc.util.validate( n.included=false, ‘‘EVENT IS NOT INCLUDED’’, [])
55
56 CALL apoc.util.validate( EXISTS((n)<-[:conditionRight]-({value:true})<-[:conditionLeft]-({
       included:true, executed:0})), ‘‘EVENT HAS A CONDITION EXPRESSION UNSATISFIED’’, [])
57
58 CALL apoc.util.validate( EXISTS((n)<-[:milestoneRight]-({value:true})<-[:milestoneLeft]-({
       included:true, pending:true})), ‘‘EVENT HAS A MILESTONE EXPRESSION UNSATISFIED’’, [])
```

To this step, the script remains almost as seen in the previous example. However, the real changes are introduced on the main trigger, as a `match` operation is a dynamic selection of a set of data-elements present on the process, being necessary to check if the data-element that is being executed belongs to one of these sets, and perform the behavior of its control relationships (remember that the main trigger is performed whenever some data-element is executed, having no data-element restriction).

With the addition of `dummy` nodes to act as relationship guards, the verification of enabledness on the script is subject to changes as well. The instructions in lines 50-54 were already present on the previous example, and verify if the node being executed is included, if all predecessors in condition relationships that are included nodes have executed in the past, and if all predecessors in milestone relationships that are included are not pending. Furthermore, the instructions in lines 56-58 guarantee the exact same thing but with the additional verification of the helper node's value, and as such, the pattern has an additional layer by splitting what was a `:condition` relationship (for e.g.) into a `:conditionLeft` and `:conditionRight`, with the verification of the dummy's value in the middle.

```
59 OPTIONAL MATCH (x_7) WHERE x_7:Y
60 OPTIONAL MATCH (dummy_8) WHERE dummy_8.reda_id = "dummy_8"
61 OPTIONAL MATCH (e_3)-[:e]->(dummy_8)
62 WITH dummy_8,x_7,e_3
63 WHERE x_7 IS NOT NULL AND e_3 IS NOT NULL AND dummy_8 IS NOT NULL
64 MERGE (x_7)-[:includesLeft]->(dummy_8)-[:includesRight]->(e_3)
```

The instruction in line 9 of Figure 4.5 is translated to the following set of instruction (lines 59-64) that are present on the main trigger: the query in line 59 fetches all data-elements that have label `Y` and gives them a local name of `x_7` – if there are some – otherwise `x_7` has the value of null but the query goes on (hence the `optional` clause). Queries in lines 60-61 retrieve the helper node alpha-renamed to `dummy_8`, as well as the node that is the correct substitution for the free name `e` in this context (note that this was the relationship created on line 35). Finally, instructions in lines 62-64 represent that if none of this local names (`x_7, e_3, dummy_8`) are null, the pattern on line 64 is going to be included into the database for every data-element belonging to the set of data-elements `x_7`.

```
65 OPTIONAL MATCH (var_6) WHERE var_6.executed>2
66 OPTIONAL MATCH (dummy_5) WHERE dummy_5.reda_id = "dummy_5"
67 OPTIONAL MATCH (b_1)-[:b]->(dummy_5)
68 WITH dummy_5,b_1,var_6
69 WHERE b_1 IS NOT NULL AND var_6 IS NOT NULL AND dummy_5 IS NOT NULL
70 MERGE (b_1)-[:relation]->(var_6)
```

The listing comprised of lines 65-70 retrieves the alpha-renamed node `dummy_5` and the set of data-elements `var_6` that have been executed more than twice, and creates a relationship named `:relation` between the data-elements belonging to this set with the correct substitution for the free name `b` in this context, which is the local name `b_1`, assuming once more that each of these local names are not null (line 69), otherwise the query aborts.

```
71 SET n.pending=false WITH n
72 OPTIONAL MATCH (n)-[:response]->(t) SET t.pending = true WITH n
73 OPTIONAL MATCH (n)-[:excludes]->(t) SET t.included = false WITH n
74 OPTIONAL MATCH (n)-[:includes]->(t) SET t.included = true WITH n
75 OPTIONAL MATCH (n)-[:responseLeft]->({value:true})-[:responseRight]->(t)
76 SET t.pending = true WITH n
77 OPTIONAL MATCH (n)-[:excludesLeft]->({value:true})-[:excludesRight]->(t)
78 SET t.included = false WITH n
79 OPTIONAL MATCH (n)-[:includesLeft]->({value:true})-[:includesRight]->(t)
80 SET t.included = true
```

Finally, to conclude the `main_trigger` content, whenever a given data-element is executed its pending property should be set to false (line 71), and the effects of the DCR inspired relationships need to be taken into account by updating the nodes that are related to the triggered node (lines 72-74), or if they have a guard, by checking if the value of the helper node that has its expression is set to `true` (lines 75-80).

Since this trigger has the verification of ReDa semantics conformance, the possible existence of several `match` operations and guards, and the application of effects of DCR inspired relationships, each of this operations is treated individually and the final result of the `main_trigger` is the union of them all, guaranteeing that each and every one of them executes despite some of them being aborted. The complete ReDa code and its respective translation to a *cypher* script is present on Annex I.

CHAPTER 5

System Architecture

The architecture of a business process management system developed to illustrate and validate the ReDa language is going to be described in this chapter. The overall system is responsible for the fully automated execution of a declarative data-driven process while providing the user the means to interact with it.

The system flow is the following: (1) it translates a fully specified ReDa process taking both data and behavior into account to its *cypher* representation; (2) the *cypher* script is instantiated on a *neo4j* database, preserving all the process semantics and its behavior in the form of database triggers; and finally, (3) the engine processes the interaction between the user and the process, providing a graphical interface coherent with its state. Additionally, we compiled the Ocaml bytecode of the compiler described on (1) to JavaScript and used it as a library on the engine (3), allowing to develop, compile, control, and interact with the process through its graphical interface.

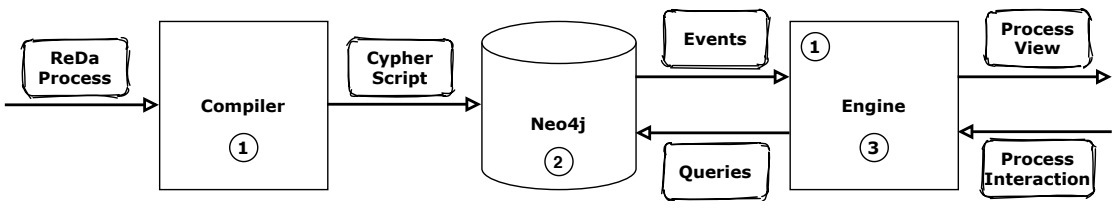

Figure 5.1: System architecture.

5.1 ReDa Compiler

Compilers are systems that translate programs written in one representation, usually of a higher-level, into an equivalent program in another form of representation more suitable for execution. The phases required for the compiler to translate ReDa processes into their equivalent *cypher* representation (the native graph language of *neo4j*) are approached here. This compiler was developed in OCaml[1], whose functional style allows for a concise and safe development [14]. We follow the usual compilation workflow where there are four major phases – lexical analysis, syntactic analysis (parsing), static-semantic validity (type checking), and code generation – used to sequentially analyze a program and synthesize a new one [19].

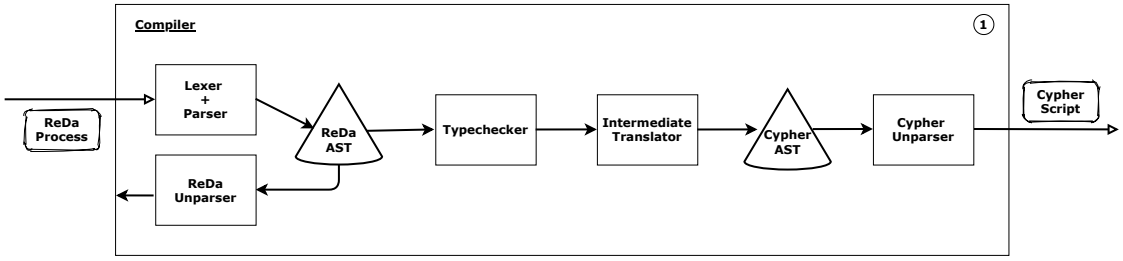

Figure 5.2: ReDa compiler architecture.

The first step in our compilation workflow is to transform the ReDa source code into an intermediate-level representation of it (ReDa Abstract Syntax Tree), a task that is carried out by both the lexer (*ocamllex*[2]) and the parser generator (*menhir*[3]) – thus accomplishing the first two phases of lexical analysis and syntactic analysis. There is also an optional step (ReDa unparser) to guarantee that both these phases preserve the original process semantics, as the original and unparsed ReDa code should be the same.

The next step comprises the static-semantic validity of the intermediate-level representation of the ReDa process. The typechecker takes as input the intermediate code and determines whether it satisfies the static-semantic properties required by ReDa, guaranteeing that errors such as the wrong declaration of identifiers or incorrect type usage (for e.g.) arise at compile time.

The final phase of our compiler is the target code generation. After the ReDa intermediate code is deemed semantically valid, the Intermediate Translator transforms it into an equivalent *cypher* intermediate code to serve as input to the *cypher* unparser, generating the *cypher* script of the initial ReDa process (the compiler applies the behavior present in Chapter 4). This final script is compatible with both *neo4j* 4.1.1 and `apoc` 4.1.0.2 [4](Section 5.2.1).

[1] https://ocaml.org
[2] https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html
[3] http://gallium.inria.fr/fpottier/menhir
[4] https://neo4j.com/developer/neo4j-apoc

5.2 Neo4j

ReDa's processes are translated into the graph-based database *neo4j* native language (*cypher*), using the database computation capabilities and reactive mechanisms (triggers) to accomplish the reactive semantics of the language. A graph database is a database management system that supports the `CRUD` – create, read, update, and delete – methods on a graph data model. They were initial built for use with transactional systems (OLTP (online transactional processing) and as so are optimized for transactional performance.

The real criteria when choosing what kind of database to use comes to what kind of data is going to be stored and how it is going to be queried. When graph databases where being proposed in the market, there must have been some strong justification for big enterprise companies like eBay, Adobe, Microsoft, IBM amongst so many others [34] to use them instead of well-established and standardized in the market `RDBMS` (relational database management systems), like the Oracle Database, MySql and so on. And as it turns out, there really is: everything can be modeled as a graph, and there is a growing demand for tailored information, meaning that the data to be shown nowadays has to be related to the user and what his interests are. Furthermore, the relationships between data can prove themselves more valuable than the data itself, stressing patterns and other valuable "hidden" information.

By using a native graph database system, where information is stored as a graph where nodes directly point to each other, the performance is improved by several orders of magnitude [26]: whereas in a relational database the connections between entities are made by inference of things like foreign keys, and the join performance degrades as the dataset gets bigger, in a graph database its core concept are the relationships themselves, and the result is a simpler and more expressive model, maintaining a relatively constant performance as the dataset grows. The aforementioned is due to connected nodes pointing to each other using index-free adjacency, and the underlying storage being designed and optimized for graphs. These kinds of databases have a significant advantage to the ones who consider themselves a graph database by exposing a graph data model through the `CRUD` operations but serializing the graph data into some relational model, as they have an additional overhead on translating the data to the correct model, thus lacking on scalability potential. In a short and concise way, a comparison can be made between `RDBMS` and graph databases:

- The `RDBMS` tables are represented as graphs;
- The rows are represented as nodes;
- `RDBMS` constraints are relationships in graph models;
- Columns and data are properties and their respective values; and finally,
- The join operation is a traversal search of the graph model.

neo4j is the world's most-used open source graph database. It is highly scalable and schema-free, and provides a flexible data model, high availability, and the use of the *cypher* query language (Section 5.2.1). Its graph data model is optimized to store, analyze, map, and traverse networks and clusters of connected information to take advantage of the value inherent in the relationships between data. It is `ACID` compliant, meaning it provides a safe environment to work, guaranteeing:

- Atomicity: either all operations in a transaction succeed or every single on of them is rolled back;
- Consistency: every operation is executed on a consistent database and its result also leaves the database in a integrity safe state;
- Isolation: ensures that multiple transactions can occur concurrently and do not contend with one another. At the logical level all operations are sequential; and finally,
- Durability: the results of a transaction are permanent and persist even in case of failure.

It also implements the most used graph data model, which is the labeled property graph model, made up of relationships and nodes with their respective labels and properties. An example of this model can be seen in Figure 5.3: two nodes are representing two distinct data elements, and their role in the graph is specified by their labels, which in this case, represent persons. A node can have multiple labels, which is extremely useful to group them into sets, and can be viewed as data storage units to properties, which are arbitrary key-value pairs. The relationships are used to connect nodes and provide structure to the graph: they can have a direction from a start node to an end node (can be the same), are identified by a name, and can have properties as well.

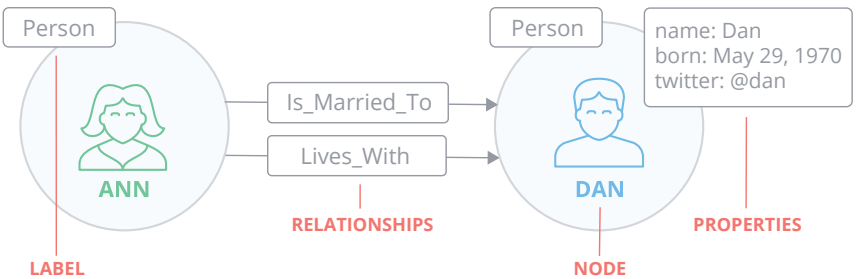

Figure 5.3: Example of a labeled property graph model[34].

neo4j provides a wide range of libraries to support user development. One of them is the `apoc` library, from *neo4j* Labs, offering support to the use of triggers and procedures, allowing the execution of a set of predetermined actions once a specific action is accomplished. This feature can be of great utility as reactivity is many times required in a database – just like it is needed to fully emulate the ReDa process language.

5.2.1 Cypher

Although there are many graph query languages nowadays, *cypher* is considered to be the standard and is the most widely deployed [9, 33]. It is inspired by both SQL and the pattern matching from SPARQL (the query language used on `RDF` triplestores, also known as semantic web), and what makes it so popular is the fact that the way *cypher* represents a graph, is the same way we intuitively describe a graph using a diagram.

As mentioned in Section 5.2, *neo4j* implements the labeled property graph model, and as such is composed of nodes, properties, values, and relationships. The *cypher* query language is based on patterns, and they are used to match the desired graph structure with the actual graph. A *cypher* query takes as an input a property graph and outputs a table, and each clause (just like the `where` clause, for e.g.) is a function that consumes a table and outputs a new table that can both add new information (tuples or number of fields) or just making modifications to it [9]. Its syntax is also simple: nodes are represented by parentheses, and their general structure is `(name:label)`, where `name` is the local name of the node variable to be identified elsewhere in the query, and `label` is the type of the node. The pattern relating nodes can be even more restrictive, constraining the node properties using key-value pairs between curly brackets:

```
(name : label { key1:value1, key2:value2, (...)})
```

It also allows the node to be specified with just the name (not restrictive, as the name is only the identifier for posterior use), only the node label (selects all nodes with that label), or the empty parentheses (meaning it is an anonymous node able to match with all nodes):

```
(name) or (:label)  or  ( )
```

The relationships pattern is also simple to understand: a pair of dashes represents an undirected relationship, and the arrowhead indicates its direction, if it has one. Relationships can also be restricted by using brackets to add details about variables, properties, or type information (just like nodes):

```
-[name : label {key : value}]-->
```

Combining the syntax of nodes and relationships it is possible to build more complex patterns. For instance, in Figure 5.3 it is possible to see a pattern that tries to match a node of type `Person` with the key-value restriction `name:Ann`, that has a relation of type `married to` with another node. The node that matches this pattern is from now on referenced as `spouse`.

The `match` clause is the heart of every *cypher* query, and it provides a specification by example, allowing the "drawing" of the pattern using `ASCII` characters. The constraints are specified in a `where` clause, and the information to be retrieved is detailed in a `return` clause. For instance:

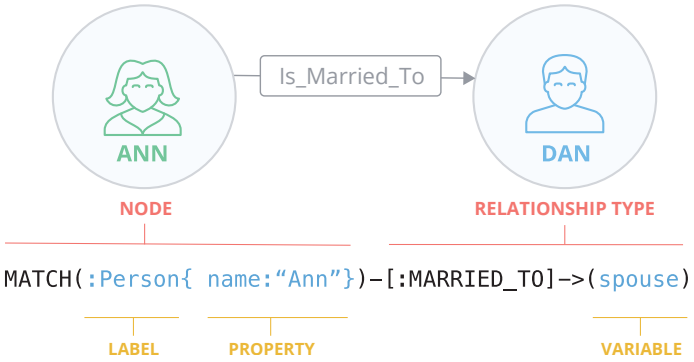

Figure 5.4: Pattern matching on *cypher*.

```
MATCH (a:Person) -[:MARRIED_TO]--> (spouse)
WHERE a.name = 'Ann'
RETURN spouse
```

There is a wide range of clauses that can also be used, like the `create` clause (creates nodes and relationships), the `union` clause (merges results from two or more queries), the `with` clause (chains subsequent query parts and forwards the result of one to another), amongst many others that are well specified in the official *neo4j* documentation [26].

5.3 ReDa Engine

A business process management system is a software able to create and manage the execution of process workflows through workflow engines. The engine is the "brain" of these systems, responsible for instantiating and controlling the execution of workflows, interpreting process definitions, providing a correct process vision to the participants and interact with them [27]. Our engine is split into two applications: one that acts as its back-end (ReDa middleware), and one that functions as its front-end (user application).

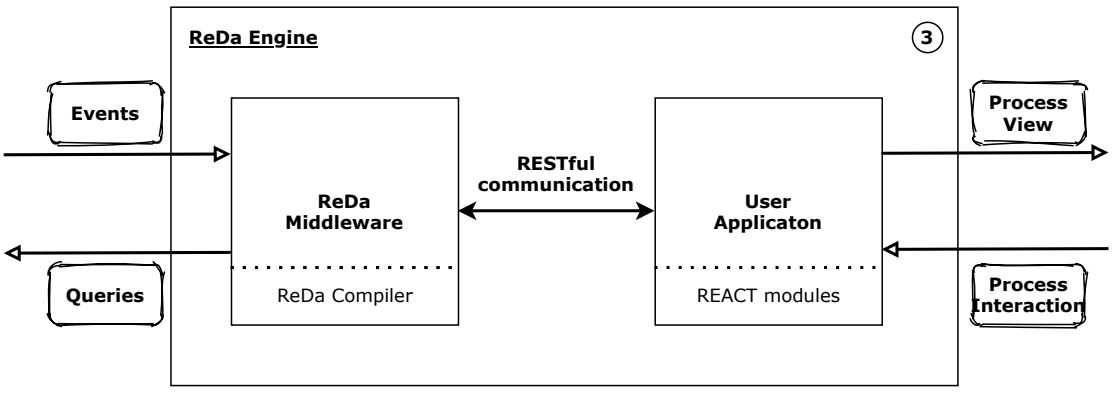

Figure 5.5: ReDa engine architecture.

The ReDa middleware – as it name suggests – bridges the gap between the user application and the database. It acts as an application programming interface disclosing the available endpoints (e.g. every input data-element or all pending data-elements), communicates with the *neo4j* database via *cypher* queries, and gets the correct events for that requests. It additionally has the ReDa compiler (Section 5.1) as a built-in service, allowing the translation of ReDa processes and its respective instantiation in the database. This service is embedded as a JavaScript file generated by `js_of_ocaml`[5], that compiles OCaml bytecode to JavaScript (due marshalling process – transforming the memory representation of an object to suit different software interfaces – is necessary to take into account). The ReDa middleware itself was developed using `NestJS`[6], a framework for building server-side applications using TypeScript.

The front-end client application of this ReDa engine was developed using `React`[7], an open-source framework supporting TypeScript. The components that form its graphical interface were individually developed and exported as `React` modules to `bit`[8] – an online UI library. The `RESTful` communication between the client-side (user application) and server-side (ReDa middleware) is made via `axios`[9], a promise-based HTTP client library.

[5] https://ocsigen.org/js_of_ocaml/3.7.0/manual/overview
[6] https://nestjs.com/
[7] https://reactjs.org/
[8] https://bit.dev/components?owners=reda_project
[9] https://www.npmjs.com/package/axios

CHAPTER 6

System Demonstration

This chapter addresses both the usage of the ReDa business process system developed and its performance. We get back to our running example of a library (where a user can create records of authors and their books in a database and manage loans to users using their names) discussed in Section 3.3, and show step-by-step the process evolution and how to interact with it. The ReDa library process is depicted in Figure 6.1.

```
1  (createAuthor:DataIN) [?:{authorName:String}]
2  ;
3  createAuthor -->> {
4    (author:DataOUT) [{name:@trigger.value.authorName}],
5    (createBook:DataIN) [?:{bookTitle:String, isbn:String}]
6    ;
7    createBook -->>{
8      (book:DataOUT) [{bookTitle:@trigger.value.bookTitle,
9                  isbn:@trigger.value.isbn, author:author.value.name}],
10     (loanBook:DataIN) [?:{username:String}]
11     ;
12     loanBook -->% loanBook,
13     loanBook -->>{
14       (loan:DataOUT) [{user:@trigger.value.username}],
15       !(returnBook:DataIN) [?]
16       ;
17       returnBook -->% returnBook,
18       returnBook -->+ loanBook
19       ;
20       loan -[:BOOK]-> book,
21       returnBook -[:LOAN]-> loan
22       }
23     ;
24     author -[:WROTE]-> book
25     }
26     ;
27 }  ;
```

Figure 6.1: ReDa library process example.

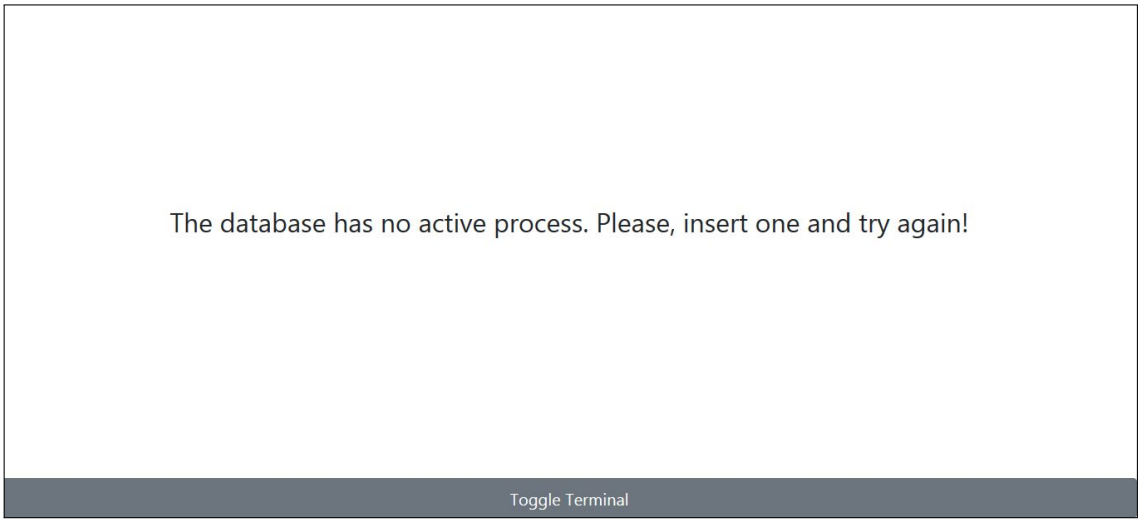

Figure 6.2: Engine's UI when there are no processes instantiated.

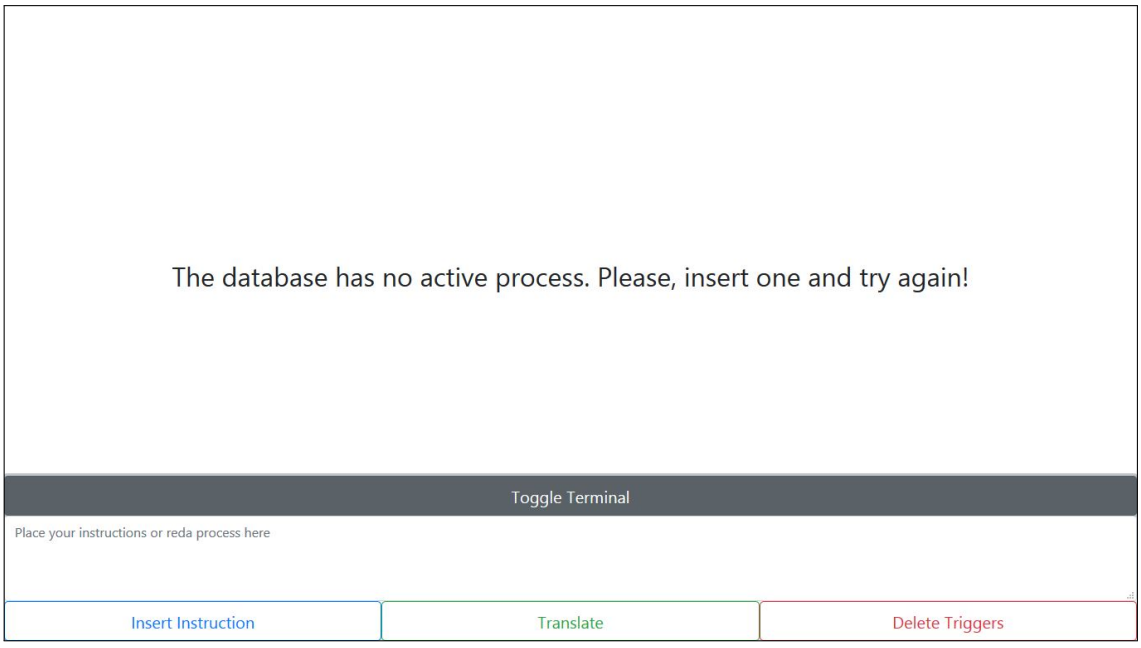

Figure 6.3: Engine's UI terminal.

Figure 6.2 shows the engine's interface when no ReDa process is instantiated, with a message conveying that same information. To interact with it there is a button at the bottom of the screen, that once pressed opens a terminal just like depicted in Figure 6.3. This terminal has three buttons: `(i)` the "Insert Instruction" button allows to insert a valid *cypher* instruction into the *neo4j* database; `(ii)` the "Translate" button compiles a ReDa process and receive its equivalent *cypher* script (it makes use of the compiler embedded in the ReDa Middleware application – Section 5.3 – via `js_of_ocaml`); `(iii)` the "Delete Triggers" button deletes all existing triggers from the database, useful for non-developers to stay away from `apoc` library calls.

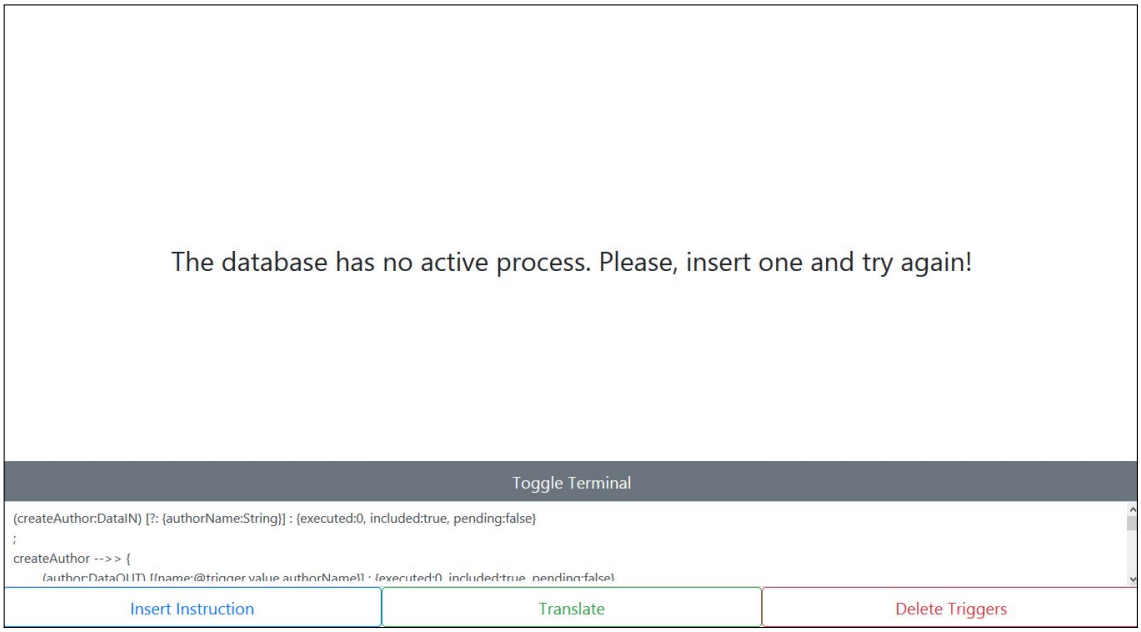

Figure 6.4: Inserting a ReDa library process for translation.

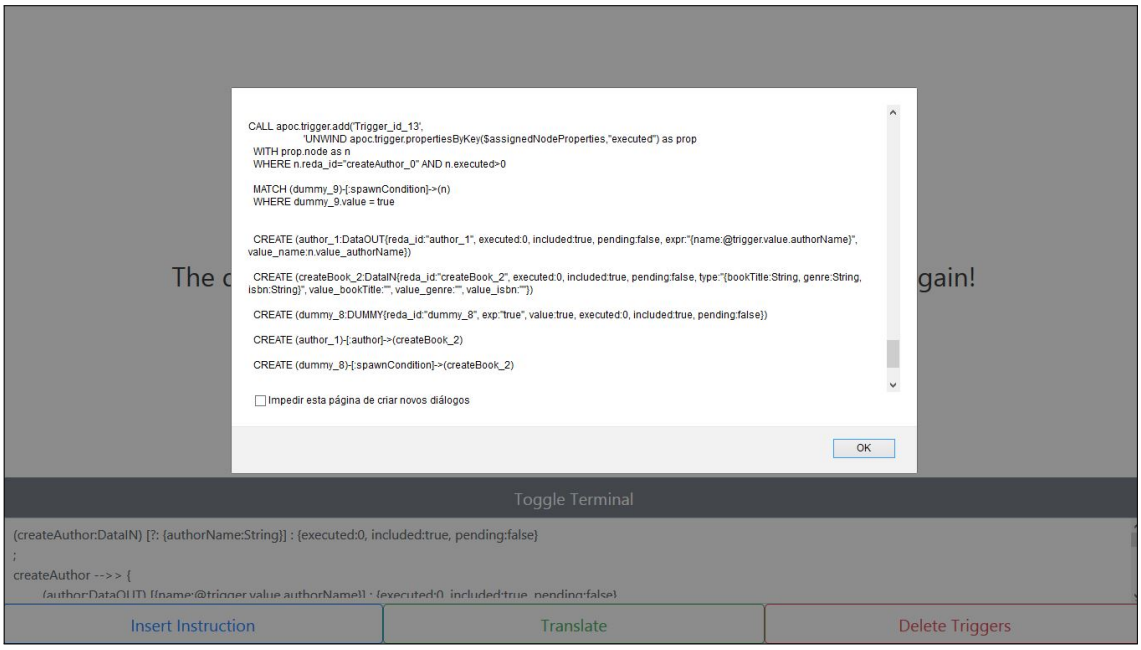

Figure 6.5: The *cypher* script translation of the ReDa library process inserted in Figure 6.4.

Getting back to our library example depicted in Figure 6.1, if we insert this ReDa process in the terminal (Figure 6.4) and then press the "Translate" button, we get its equivalent compiled *cypher* script (Figure 6.5, also present on Annex III). The next step consists on the insertion of the process into the database: this is achieved by pressing "Insert Instruction" to the instructions of the obtained *cypher* script, leading us to the screen of Figure 6.6.

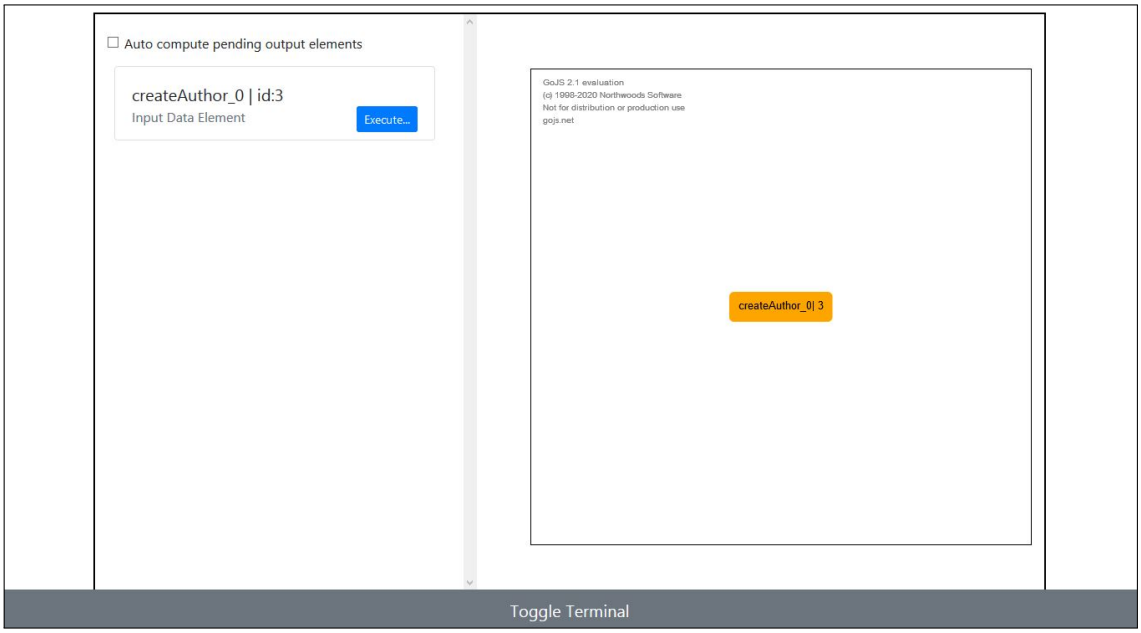

Figure 6.6: Engine's UI state after the instantiation of the library ReDa process.

Figure 6.6 depicts the state of the user interface after the process insertion. There is only one data-element in the process workflow at this step (`createAuthor`) present at the left side of the screen, where all existent data-elements are listed. Each of these listed nodes has its card component, with: `(i)` its alpha-renamed local identifier and database identifier on top; `(ii)` its grayed-out label on the bottom; `(iii)` a blue `execute...` button on the card's right side, which when pressed displays all of the said data-element properties and its respective values, and, in the case of an input node, a form with its input fields; and finally, `(iv)` the card border and background color to specify if a node is pending (red border) or excluded (gray background and without the `execute...` button) – the `createAuthor` data-element is both included and not pending, and has such has the default card. Still on the left side of the screen but now at its top, we have a check-box to "Auto compute pending output elements", meaning that when ticked all pending output data-elements are automatically executed by the system. To conclude, at the right side of the screen we have a visual representation of the process graph, updated at each execution that may cause it to change, where each node is identified by its alpha-renamed identifier and database identifier.

If we now press the `createAuthor`'s "`execute...`" button, the modal of Figure 6.7 with all its properties and respective values is displayed. As an input data-element with type record, there is also a form to insert the `authorName` that when executed with "Tolkien" as a value causes the system UI to render the screen of Figure 6.8. Here there are two new data-elements: the `Tolkien` and its `createBook` data-element (notice that the graph that represents the process state has an implicit `:author` relationship between these elements, as the author data-element is a free name inside the spawn of `createBook` – defined in line 9 of Figure 6.1).

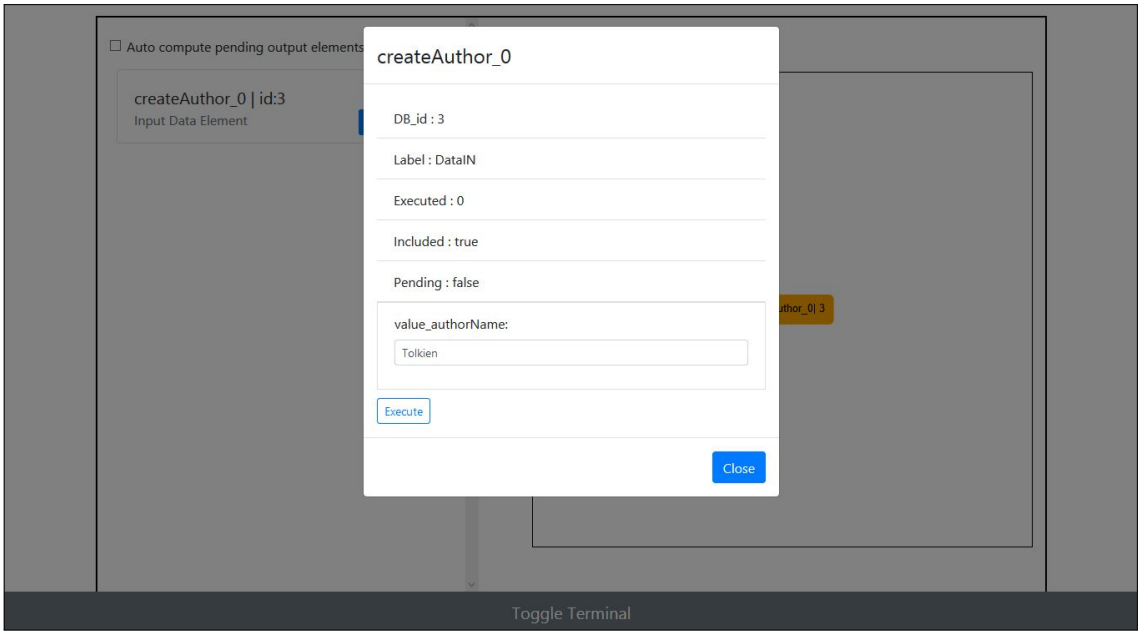

Figure 6.7: Modal referring to the input data-element createAuthor.

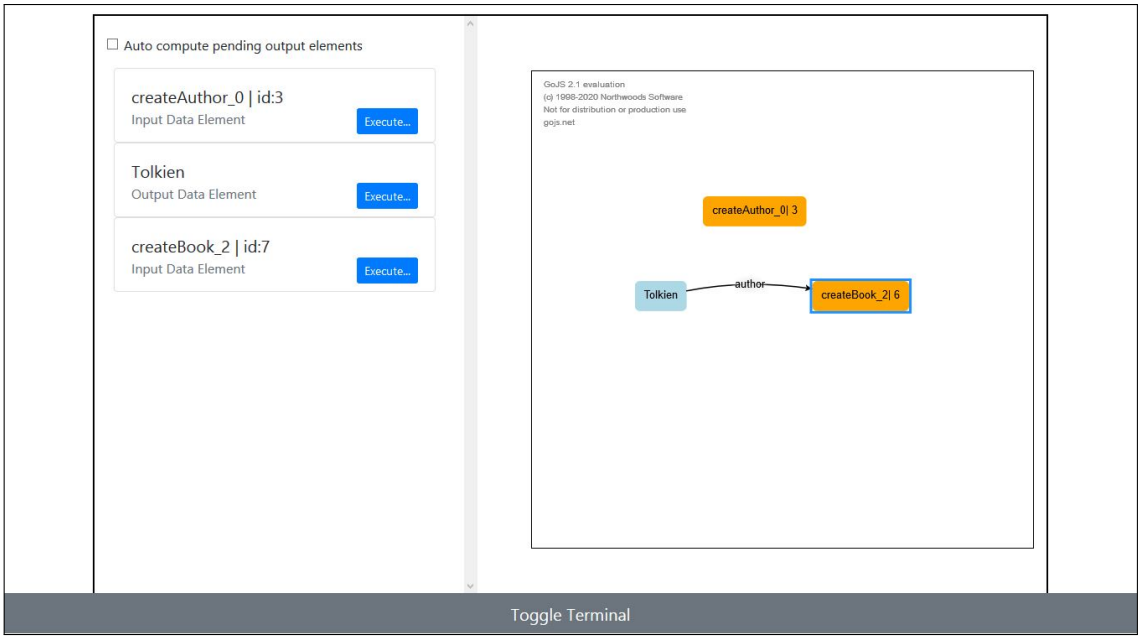

Figure 6.8: Process state after the execution of the createAuthor input data-element.

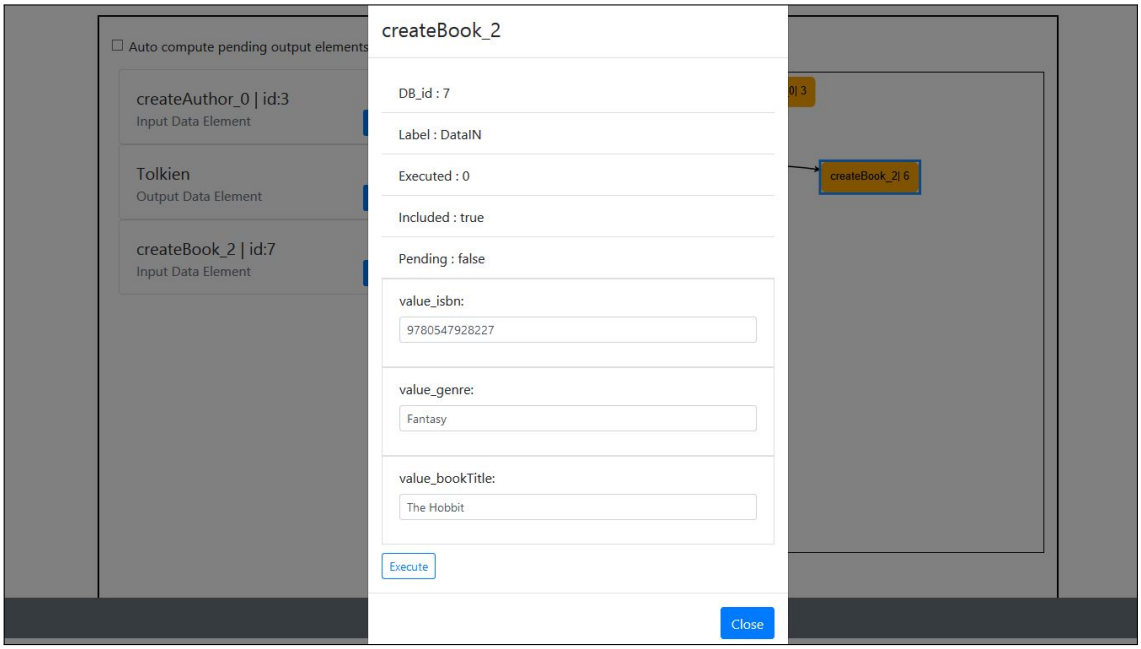

Figure 6.9: Modal referring to the input data-element createBook of the Tolkien author.

Continuing the process workflow, if the input data-element `createBook` is executed with "The Hobbit" as a value for the `bookTitle`, "Fantasy" as a value for the `genre`, and "97805479288277" as a value for the `isbn` (Figure 6.9), the screen of Figure 6.10 is displayed with the graph that represents the process state having:

- two new data-elements: `The Hobbit` output data-element and the `loanBook` input data-element, as specified on lines 4 and 5 of Figure 6.1;
- the explicit data relationship `:WROTE` between the output data-element `Tolkien` and the output data-element `The Hobbit` (specified on line 24 of the library process Figure 6.1), as well as the implicit relationships `:author`, `:response`, and `:condition`, stating that only and whenever the `Tolkien` data-element is executed the output data-element `The Hobbit` is set to pending, as `author` is a free name in its properties (line 9) – therefore a dependency – and its occurrences need to be evaluated to the data-element `Tolkien`;
- the implicit `:book` relationship between the input data-element `loanBook` and the output data-element `The Hobbit`, as book is a free name in the sub-process of `loanBook` (line 20 of Figure 6.1), and this relation guarantees that every occurence of the free name `book` inside the scope of the sub-process created in the execution of `loanBook` is evaluated to `The Hobbit`; and finally,
- the `loanBook` data-element also has an explicit `:excludes` relationship with itself (line 12 of Figure 6.1), meaning that when it is executed it excludes itself from the process.

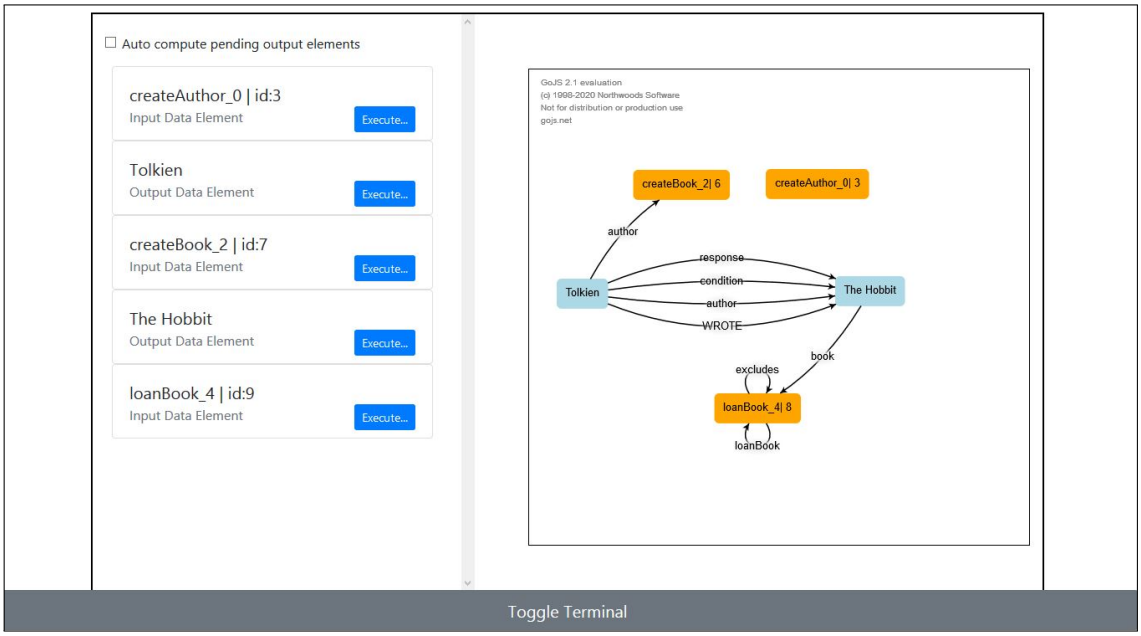

Figure 6.10: Process state after the execution of the createBook input data-element.

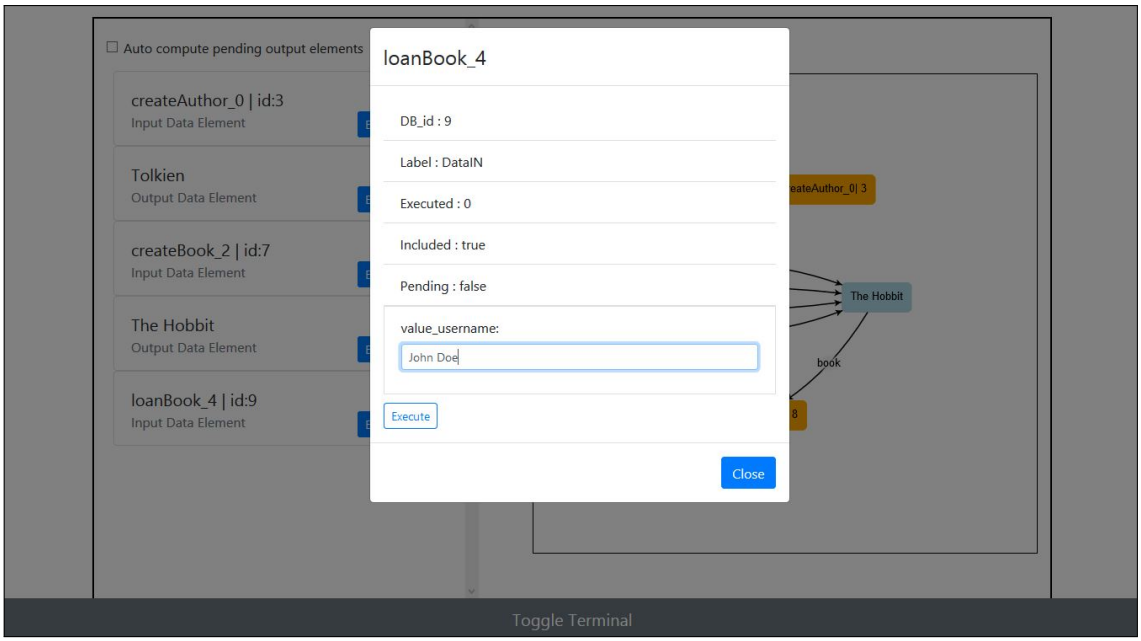

Figure 6.11: Modal referring to the input data-element loanBook.

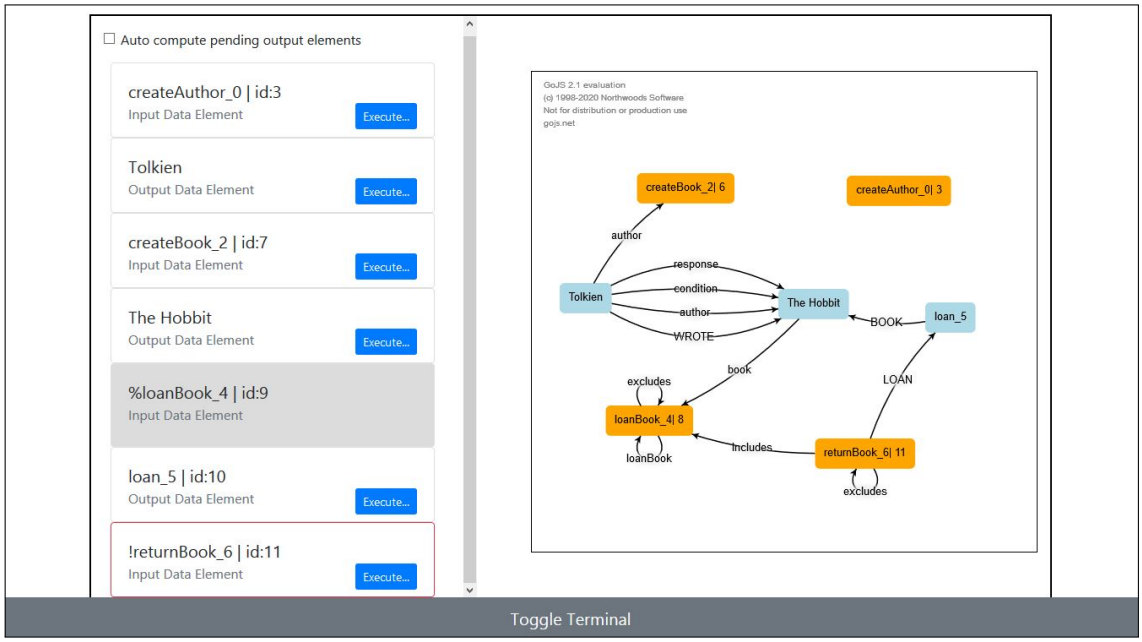

Figure 6.12: Process state after the execution of the loanBook input data-element.

If we now want to loan a book in the library, we do so by executing `loanBook` (Figure 6.11) with "John Doe" as an argument (for e.g.), causing the engine to render the screen of Figure 6.12 where the remainder of the explicit relationships defined in Figure 6.1 are added (lines 17-18 and 20-21), as well as a new `loan` output data-element with informations concerning the loaned book (`The Hobbit` in this case) and the name of the loaner, and a pending input data-element `returnBook` for that loan (listed with a red border card component on the left side of the screen). This action also causes the `loanBook` data-element to exclude itself from the process, reflected in the list of data-elements as the `loanBook` has now the excluded card component (gray background without the button to be executed).

Next, if we execute the `returnBook` input data-element (the screen depicted in Figure 6.13 – notice that it is an input data-element with type `unit` having no required information to be submitted) it excludes itself from the process and includes once again the previously excluded `loanBook` data-element, allowing the book `The Hobbit` to be loaned again (as depicted in Figure 6.14). As there are no pending data-elements, and having seen the effect of every possible action in the process, we can terminate our process execution here.

Notice also that this is just an example of a possible workflow for this process using the context of one author and the sub-context of one book, as it was also possible to have a workflow with one author and n books, each with its `loanBook` input data-element and therefore being independent from each other, or even n authors and n books, and interchangeably loan and return different books from the same author or different authors.

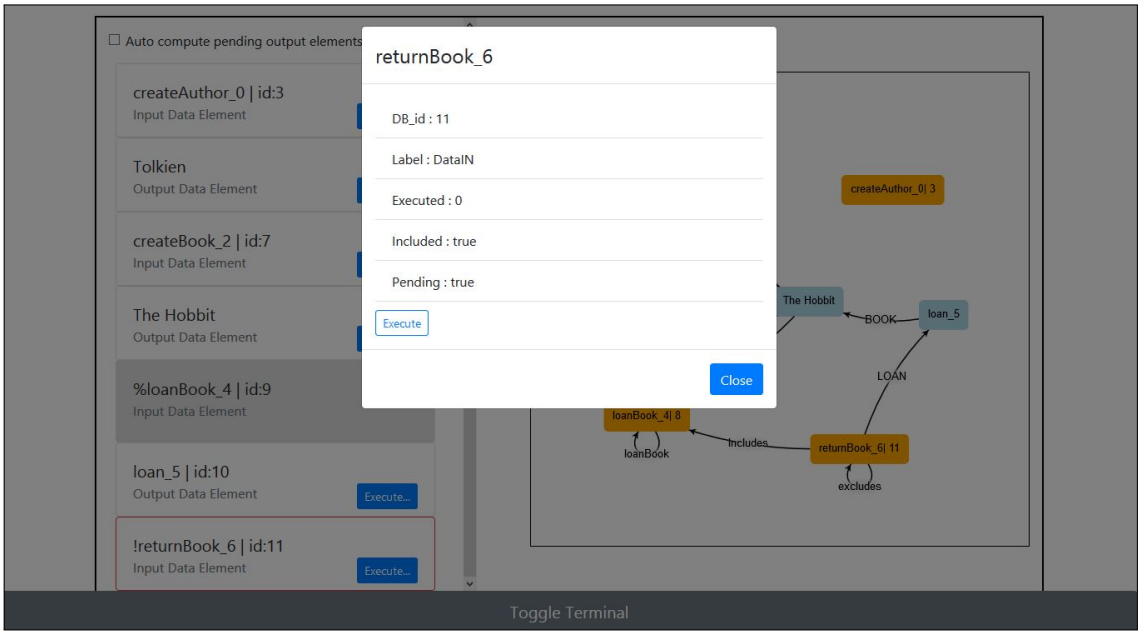

Figure 6.13: Modal referring to the input data-element returnBook.

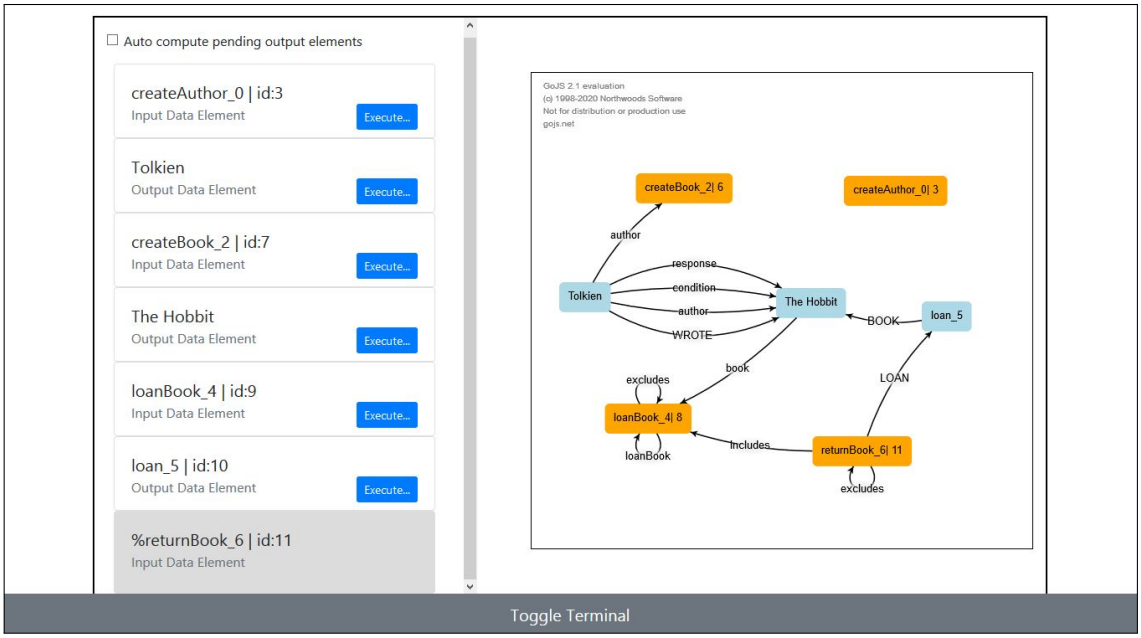

Figure 6.14: Process state after the execution of the returnBook input data-element.

6.1 Performance

We tested the communication time between the database and the engine in scenarios where there are thousands of nodes and relationships between them to measure the scalability potential of our prototype (note that the database triggers are static, and therefore their number remains constant throughout all execution). We used two case studies for this purpose: one that represents a so-called "normal" program, and another that consists of a worst-case scenario. These programs are purposely developed to grow linearly in twenty-five executions, each performed ten times. These case studies were executed in a database *neo4j* 4.1.1 with APOC 4.1.0.2, and the engine hosted on Firefox Developer Edition 83.0, in a Windows 8.1, Intel Core i7-4510U CPU @ 2.00GHz 2.60GHz with 8Gb RAM system.

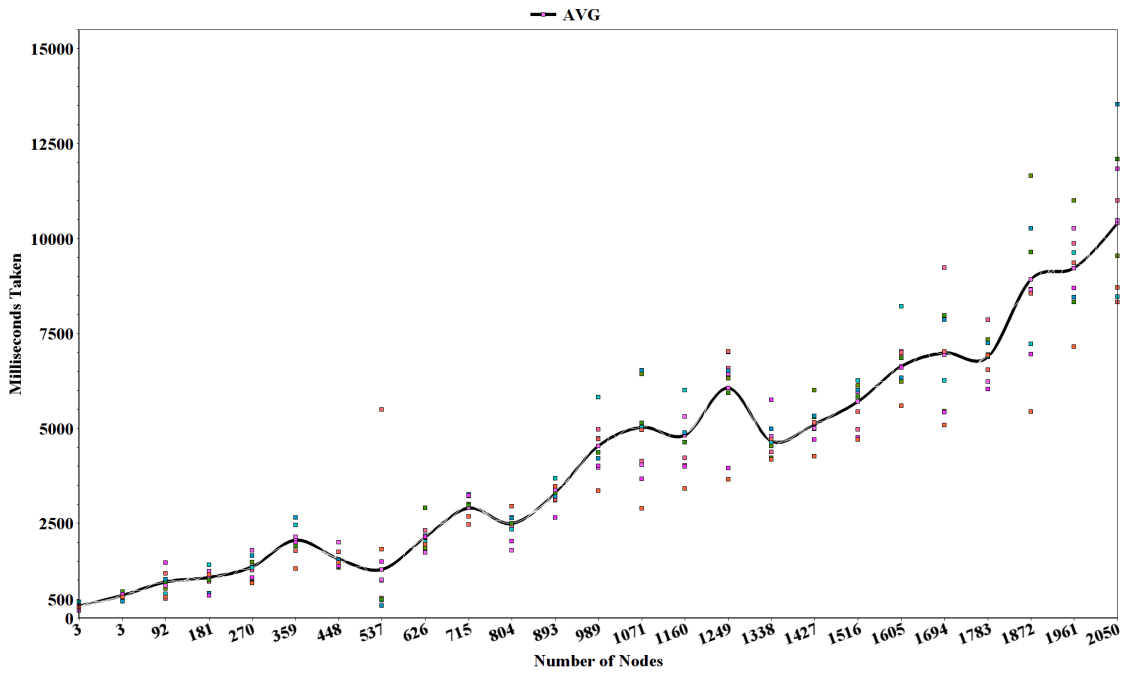

Figure 6.15: Normal-case scenario performance.

```
1  (a:X)[3],
2  (d:X)[a.value]
3  ;
4  d -->> {
5    (87 nodes split into labels X, Y and Z are created here)
6    ;
7    ;
8    (MATCH (var) WHERE var?Y RETURN var) -[:rel1]-> (MATCH (var) WHERE var?X RETURN var)
9  }
10 ;
```

Figure 6.16: ReDa process used to perform the case study of a normal-case scenario.

The "normal" case scenario consists of a program where nodes and relationships are split amongst different clusters. As present in Figure 6.16, this case study starts with only two data-elements belonging to one cluster (line 1-2, cluster X), with each further execution of the data-element `d` adding eighty-seven new data-elements that are split almost equally into three clusters (X, Y and Z). Each data-element has either no relationships (spawner input data-elements, label Z), or has between `one` to `a third of the existent data-elements` relationships (line 8 states that each element of cluster Y has a relationship `:rel1` with all elements of cluster X), as they also tend to grow alongside the program.

As depicted in Figure 6.15, the time difference between the request and the response tends to grow in a somewhat linear to sub-linear fashion with each execution.

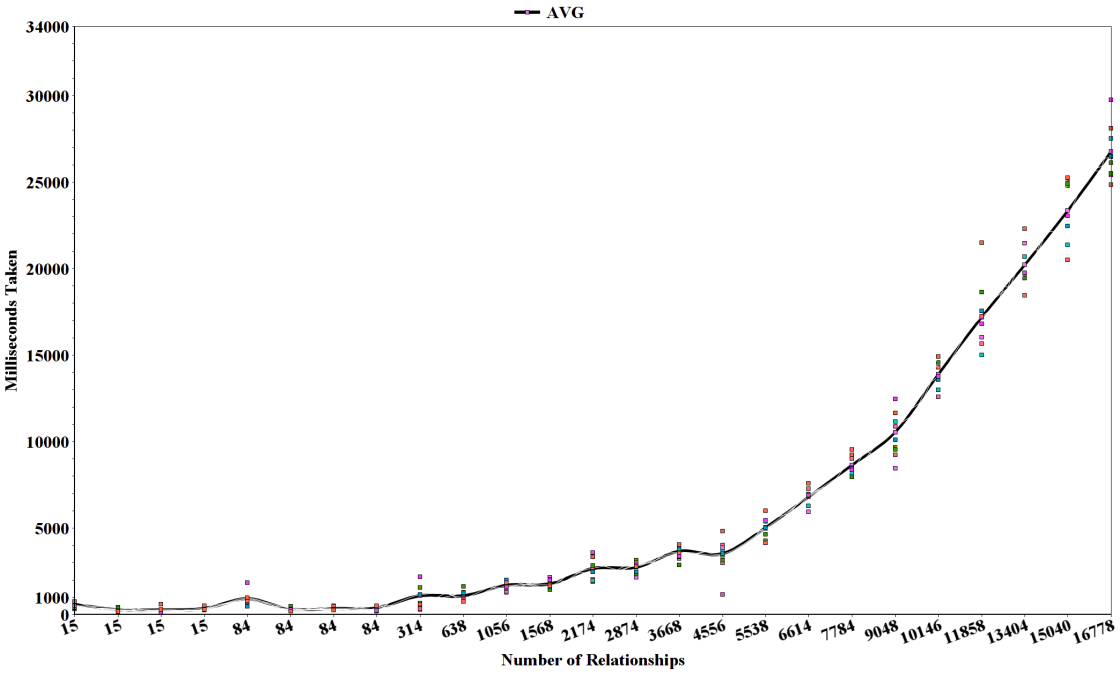

Figure 6.17: Worst-case scenario performance.

However, the same thing does not happen when dealing with the worst-case scenario (Figure 6.18). Here, we have three different process levels: `(i)` the outer context with data-elements of label `X`, and where `a` is the single data-element in the whole process that does not have a dependency on any other data-element, as `b` depends on `a`, `c` depends on `b`, and `d` depends on both `b` and `c` (lines 1-4); `(ii)` whenever `d` is executed a new sub-process with data-elements of label `Y` dependent on all the data-elements of label `X` are included (lines 7-10), as well as an explicit relationship between them (line 23); `(iii)` whenever `h` is executed a new sub-process with data-elements of label `Z` dependent on all the data-elements of label `X` and `Y` are included (lines 13-16), as well as an explicit relationship between elements of label `Z` and every other data-element (themselves included, line 20), and a response relationship between every data-element (with an aggregation guard that

evaluates always to `true` but adds an overhead to process it, line 18).

This means that we have only one giant cluster where data-elements are highly dependent on each other (the number of data-elements here ranges between 4 in the first execution to 113 in the last, growing linearly), and the time difference between the request and the response tends to grow in a linear to super-linear fashion, as depicted in Figure 6.17.

```
1  (a:X)[3],
2  (b:X)[a.value],
3  (c:X)[b.value],
4  (d:X)[c.value + b.value]
5  ;
6  d -->> {
7    (e:Y)[a.value+b.value+c.value+d.value],
8    (f:Y)[a.value+b.value+c.value+d.value],
9    (g:Y)[a.value+b.value+c.value+d.value],
10   (h:Y)[a.value+b.value+c.value+d.value]
11   ;
12   h -->> {
13     (i:Z)[a.value+b.value+c.value+d.value+e.value+f.value+g.value+h.value],
14     (j:Z)[a.value+b.value+c.value+d.value+e.value+f.value+g.value+h.value],
15     (k:Z)[a.value+b.value+c.value+d.value+e.value+f.value+g.value+h.value],
16     (l:Z)[a.value+b.value+c.value+d.value+e.value+f.value+g.value+h.value]
17     ;
18     (MATCH (var) WHERE true RETURN var) *-[MATCH (var) WHERE true WITH COUNT(var) AS count
           WHERE count>0]-> (MATCH (var) WHERE true RETURN var)
19     ;
20     (MATCH (var) WHERE var?Z RETURN var) -[:rel2]-> (MATCH (var) WHERE true RETURN var)
21   }
22   ;
23   (MATCH (var) WHERE var?Y RETURN var) -[:rel1]-> (MATCH (var) WHERE var?X RETURN var)
24 }
25 ;
```

Figure 6.18: ReDa process used to perform the case study of a worst-case scenario.

To conclude, triggers are many times sources of performance issues in database systems. With this in mind, we made every effort to encode all the reactivity and behavior of a ReDa process into static triggers, remaining constant in number throughout all execution. There are in maximum two triggers activated each time a specific data-element is executed: `(i)` the trigger regarding that specific data-element, containing the specific behavior defined in the process; and `(ii)` the `main_trigger` that is always activated despite the data-element being executed, containing the `enabledness` verification, the `match` and `guard` verification (to see if the data-element being executed belongs to the set of data-elements selected by the match clause, or if the guard of a dynamic relationship changes its value), and the application of DCR effects (like the inclusion relationship, for instance). With this information and the results of our case study, we can infer that the main cause for performance deterioration is the number of control relationships that the node being executed has, as it is necessary to analyze each of them in the `main_trigger` to have the next ReDa process state.

CHAPTER 7

Conclusions

This dissertation introduced ReDa, a declarative process definition and programming language for describing REactive DAta-driven processes, where data is intuitively stored in a graph-based structure to define explicit and implicit relations between data-elements, generalizing both the relational and the semi-structured data model used in related approaches [3]. We also presented a compilation procedure to systematically transform a ReDa process into a *cypher* script to be used by our prototype of a business process system, embedding the computations and control-flow defined by the process using the native capabilities of a standalone graph-database *neo4j* and the ReDa middleware.

The proposal of this thesis meant to work on and improve a draft of a process language that would generalize ReSeDa – what is now the ReDa process language – and develop a compiler that would translate it to be instantiated in a graph-database, while fully preserving its semantics and reactive properties. As our initial goals were being accomplished we sought more: a business process system that could emulate said processes and provide interaction with the user would be a way to measure its applicability. It is safe to say that our ambitions are met.

Another positive remark is what can follow this line of work: the next steps can include the definition of a bidirectional procedure that allows the live edit of ReDa processes while verifying the integrity of existing data and control upon the introduction of new process elements, and the addition of user roles in the process language, restricting events and providing a notion of hierarchy. Another interesting path to follow is to explore the work on reactors [22] (application-defined OLTP databases) as a possible target language for ReDa instead of *neo4j*.

Bibliography

[1] D. Basin, S. Debois, and T. T. Hildebrandt. "In the nick of time: Proactive prevention of obligation violations." In: *Proceedings - IEEE Computer Security Foundations Symposium* 2016-Augus (2016). ISSN: 19401434. DOI: 10.1109/CSF.2016.16.

[2] E. Börger. "Approaches to modeling business processes: A critical analysis of BPMN, workflow patterns and YAWL." In: *Software and Systems Modeling* 11.3 (2012), pp. 305–318. ISSN: 16191366. DOI: 10.1007/s10270-011-0214-z.

[3] J. Costa Seco, S. Debois, T. Hildebrandt, and T. Slaats. "RESEDA: Declaring live event-driven computations as reactive semi-structured data." In: *Proceedings - 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference, EDOC 2018* May (2018), pp. 75–84. DOI: 10.1109/EDOC.2018.00020.

[4] S. Debois, T. Hildebrandt, and T. Slaats. "Hierarchical declarative modelling with refinement and sub-processes." In: *International Conference on Business Process Management*. Springer, Cham. 2014, pp. 18–33.

[5] S. Debois, T. Hildebrandt, and T. Slaats. "Hierarchical declarative modelling with refinement and sub-processes." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8659 LNCS.September (2014), pp. 18–33. ISSN: 16113349. DOI: 10.1007/978-3-319-10172-9_2.

[6] S. Debois, T. Hildebrandt, and T. Slaats. "Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9109.grant 33295 (2015), pp. 143–160. ISSN: 16113349. DOI: 10.1007/978-3-319-19249-9_10.

[7] S. Debois, T. T. Hildebrandt, and T. Slaats. "Replication, Refinement & Reachability: Complexity in Dynamic Condition-Response Graphs." en. In: *Acta Informatica* 55.6 (Sept. 2018), pp. 489–520. ISSN: 0001-5903, 1432-0525. DOI: 10.1007/s00236-017-0303-8.

[8] R. Eshuis, S. Debois, T. Slaats, and T. Hildebrandt. "Deriving consistent GSM schemas from DCR graphs." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*

9936 LNCS (2016), pp. 467–482. ISSN: 16113349. DOI: 10.1007/978-3-319-46295-0_29.

[9] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. "Cypher: An Evolving Query Language for Property Graphs." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657.

[10] L. Galrinho, J. Costa Seco, S. Debois, T. Hildebrandt, and T. Slaats. "ReDa: Reactive Data-driven Processesas Graph Databases." In: *DEC2H 2020 : 8th International Workshop on DEClarative, DECision and Hybrid approaches to processes* (2020).

[11] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, and G. Wirtz. "BPMN conformance in open source engines." In: *Proceedings - 9th IEEE International Symposium on Service-Oriented System Engineering, IEEE SOSE 2015* 30 (2015), pp. 21–30. DOI: 10.1109/SOSE.2015.22.

[12] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz. "BPMN 2.0: The state of support and implementation." In: *Future Generation Computer Systems* 80 (2018), pp. 250–262. ISSN: 0167739X. DOI: 10.1016/j.future.2017.01.006. URL: http://dx.doi.org/10.1016/j.future.2017.01.006.

[13] Y. B. Han, J. Y. Sun, G. L. Wang, and H. F. Li. "A cloud-based BPM architecture with user-end distribution of non-compute-intensive activities and sensitive data." In: *Journal of Computer Science and Technology* 25.6 (2010), pp. 1157–1167. ISSN: 10009000. DOI: 10.1007/s11390-010-9396-z.

[14] J. Hickey, A. Madhavapeddy, and Y. Minsky. *Real World OCaml*. 2014. ISBN: 144932391. URL: http://www.worldcat.org/isbn/144932391.

[15] T. Hildebrandt and R. R. Mukkamala. "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs." In: *Post-Proceedings of PLACES 2010*. Vol. 69. EPTCS. 2010, pp. 59–73. DOI: 10.4204/EPTCS.69.5.

[16] T. T. Hildebrandt and R. R. Mukkamala. "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs." In: *Electronic Proceedings in Theoretical Computer Science* 69 (2011), pp. 59–73. ISSN: 2075-2180. DOI: 10.4204/eptcs.69.5.

[17] G. Li, R. M. de Carvalho, and W. M. P. van der Aalst. "Object-centric behavioral constraint models." In: (2019), pp. 48–56. DOI: 10.1145/3297280.3297287. arXiv: 1703.05740.

[18] H. Mili, G. Tremblay, G. B. Jaoude, É. Lefebvre, L. Elabed, and G. E. Boussaidi. "Business process modeling languages: Sorting through the alphabet soup." In: *ACM Computing Surveys* 43.1 (2010). ISSN: 03600300. DOI: 10.1145/1824795.1824799.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation*. 1997. ISBN: 1558603204.

[20] M. Pesic, H. Schonenberg, and W. M. Van Der Aalst. "DECLARE: Full support for loosely-structured processes." In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC* May 2014 (2007), pp. 287–298. ISSN: 15417719. DOI: 10.1109/EDOC.2007.4384001.

[21] P. Sestoft. *Spreadsheet implementation technology. Basics and extensions*. United States: MIT Press, 2014. ISBN: 978-0-262-52664-7.

[22] V. Shah and M. A. Vaz Salles. "Reactors: A Case for Predictable, Virtualized Actor Database Systems." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. 2018, 259–274. ISBN: 9781450347037. DOI: 10.1145/3183713.3183752. URL: https://doi.org/10.1145/3183713.3183752.

[23] M. Skouradaki, D. H. Roller, F. Leymann, V. Ferme, and C. Pautasso. "On the road to benchmarking BPMN 2.0 workflow engines." In: *ICPE 2015 - Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (2015), pp. 301–304. DOI: 10.1145/2668930.2695527.

[24] J. Su, L. Wen, and J. Yang. "From Data-centric Business Processes to Enterprise Process Frameworks." In: *Proceedings - 2017 IEEE 21st International Enterprise Distributed Object Computing Conference, EDOC 2017* 2017-Janua.October (2017), pp. 1–9. DOI: 10.1109/EDOC.2017.11.

[25] W. Van Der Aalst, A. Artale, M. Montali, and S. Tritini. "Object-centric behavioral constraints: Integrating data and declarative process modelling." In: *CEUR Workshop Proceedings* 1879 (2017). ISSN: 16130073.

[26] J. Webber and E. Eifrem. *Graph Databases*. Second Edi. O'Reilly, 2015. ISBN: 9781491930892.

[27] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. 3rd ed. Springer Berlin Heidelberg, 2019. ISBN: 978-3-662-59431-5;978-3-662-59432-2. DOI: 10.1007/978-3-642-28616-2.

Webography

[28] AIIM. *What is Business Process Management?* 2020. URL: https://www.aiim.org/What-is-BPM (visited on 01/17/2020).

[29] I. K. Center. *Business Process BPML Components*. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SS3JSW_5.2.0/com.ibm.help.bpml.doc/SI_BusinessProcessBPMLComponents.html (visited on 01/17/2020).

[30] C. S. GmbH. *Workflow and Decision Automation Platform | Camunda BPM*. 2020. URL: https://camunda.com/ (visited on 01/18/2020).

[31] JBPM. *jBPM - Open Source Business Automation Toolkit - jBPM Business Automation Toolkit*. 2020. URL: https://www.jbpm.org/ (visited on 01/19/2020).

[32] Kissflow. *What Is a Workflow Process?* 2019. URL: https://kissflow.com/workflow/workflow-vs-process-whats-difference/ (visited on 01/17/2020).

[33] Neo4j. *Cypher Graph Query Language*. 2020. URL: https://neo4j.com/cypher-graph-query-language/ (visited on 01/24/2020).

[34] Neo4j. *Neo4j Graph Platform – The Leader in Graph Databases*. 2020. URL: https://neo4j.com/ (visited on 01/23/2020).

[35] Object Management Group. *BPMN — Open Issues - OMG Issue Tracker*. 2020. URL: https://issues.omg.org/issues/spec/BPMN/2.0 (visited on 02/03/2020).

[36] N. Palmer. *What is BPM?* 2014. URL: https://bpm.com/what-is-bpm (visited on 01/17/2020).

ANNEX I

ReDa Process Translation Example

I.1 ReDa Process

```
(a:X)[:?],
(b:X)[3],
(c:X)[true],
%(e:Y)[true]
;
a -[b.value>0]->> {
  (d:Y)[true]
  ;
  (MATCH (x) WHERE x?Y RETURN x) -[c.value]->+ e
  ;
  b -[:relation]-> MATCH (var) WHERE var.executed>2 RETURN var
}
;
```

I.2 Cypher Script

```
CREATE (a_0:X{reda_id:"a_0", executed:0, included:true, pending:false, value:null})
CREATE (b_1:X{reda_id:"b_1", executed:0, included:true, pending:false, value:3})
CREATE (c_2:X{reda_id:"c_2", executed:0, included:true, pending:false, value:true})
CREATE (e_3:Y{reda_id:"e_3", executed:0, included:false, pending:false, value:true})
CREATE (dummy_9:DUMMY{reda_id:"dummy_9", exp:"b.value>0", executed:0, included:true,
    pending:false})
CREATE (b_1)-[:response]->(dummy_9)
CREATE (dummy_9)-[:spawnCondition]->(a_0)
CREATE (b_1)-[:b]->(a_0)
CREATE (e_3)-[:e]->(a_0)


CALL apoc.trigger.add(‘‘Main Trigger’’,
```

```
13 ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
14  WITH prop.node as n WHERE n.executed > 0
15
16  CALL apoc.util.validate( EXISTS((n)<- [:condition] - ({included:true, executed:0})), ‘‘
        EVENT HAS A CONDITION UNSATISFIED’’, [])
17  CALL apoc.util.validate( EXISTS((n)<- [:milestone] -({included:true, pending:true})), ‘‘
        EVENT HAS A MILESTONE UNSATISFIED’’, [])
18  CALL apoc.util.validate( n.included=false, ‘‘EVENT IS NOT INCLUDED’’, [])
19  CALL apoc.util.validate( EXISTS((n)<- [:conditionRight] -({value:true})<- [:
        conditionLeft] -({included:true, executed:0})), ‘‘EVENT HAS A CONDITION EXPRESSION
        UNSATISFIED’’, [])
20  CALL apoc.util.validate( EXISTS((n)<- [:milestoneRight] -({value:true})<- [:
        milestoneLeft] -({included:true, pending:true})), ‘‘EVENT HAS A MILESTONE EXPRESSION
         UNSATISFIED’’, [])
21
22  RETURN 1 as X
23
24  UNION
25
26  UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
27  WITH prop.node as n WHERE n.executed > 0
28
29  OPTIONAL MATCH (x_7) WHERE x_7:Y
30  OPTIONAL MATCH (dummy_8) WHERE dummy_8.reda_id = "dummy_8"
31  OPTIONAL MATCH (e_3)-[:e]->(dummy_8)
32  WITH dummy_8,x_7,e_3
33  WHERE x_7 IS NOT NULL AND e_3 IS NOT NULL AND dummy_8 IS NOT NULL
34  MERGE (x_7)-[:includesLeft]->(dummy_8)-[:includesRight]->(e_3)
35
36  RETURN 1 as X
37
38  UNION
39
40  UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
41  WITH prop.node as n WHERE n.executed > 0
42
43  OPTIONAL MATCH (var_6) WHERE var_6.executed>2
44  OPTIONAL MATCH (dummy_5) WHERE dummy_5.reda_id = "dummy_5"
45  OPTIONAL MATCH (b_1)-[:b]->(dummy_5)
46  WITH dummy_5,b_1,var_6
47  WHERE b_1 IS NOT NULL AND var_6 IS NOT NULL AND dummy_5 IS NOT NULL
48  MERGE (b_1)-[:relation]->(var_6)
49
50  RETURN 1 as X
51
52  UNION
53
54  UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
55  WITH prop.node as n WHERE n.executed>0
56
```

```
57    SET n.pending=false WITH n
58    OPTIONAL MATCH (n)-[:response]->(t) SET t.pending = true WITH n
59    OPTIONAL MATCH (n)-[:excludes]->(t) SET t.included = false WITH n
60    OPTIONAL MATCH (n)-[:includes]->(t) SET t.included = true WITH n
61    OPTIONAL MATCH (n)-[:responseLeft]->({value:true})-[:responseRight]->(t)
62    SET t.pending = true WITH n
63    OPTIONAL MATCH (n)-[:excludesLeft]->({value:true})-[:excludesRight]->(t)
64    SET t.included = false WITH n
65    OPTIONAL MATCH (n)-[:includesLeft]->({value:true})-[:includesRight]->(t)
66    SET t.included = true
67
68    RETURN 1 as X
69  ',{phase:``before''});
70
71
72  CALL apoc.trigger.add(``When a_0 happens'',
73   `UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
74    WITH prop.node as n WHERE n.reda_id="a_0" AND n.executed>0
75
76    MATCH (b_1)-[:b]->(n)
77    MATCH (e_3)-[:e]->(n)
78    MATCH (dummy_9)-[:spawnCondition]->(n) WHERE dummy_9.value = true
79
80    CREATE (d_4:Y{reda_id:"d_4", executed:0, included:true, pending:false, value:true})
81    CREATE (dummy_5:DUMMY{reda_id:"dummy_5", exp:"true", value:true, executed:0, included:
          true, pending:false})
82    CREATE (dummy_8:DUMMY{reda_id:"dummy_8", exp:"c.value", executed:0, included:true,
          pending:false})
83
84    CREATE (b_1)-[:b]->(dummy_5)
85    CREATE (c_2)-[:response]->(dummy_8)
86    CREATE (e_3)-[:e]->(dummy_8)
87
88    RETURN 1 as X
89  ',{phase:``before''});
90
91
92  CALL apoc.trigger.add(``When dummy_9 happens'',
93   `UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,``executed'') as prop
94    WITH prop.node as n WHERE n.reda_id=``dummy_9'' AND n.executed>0
95
96    MATCH (b_1)-[:response]->(n)
97    WHERE b_1.reda_id = "b_1"
98
99    SET n.value = b_1.value > 0
100
101   RETURN 1 as X
102 ',{phase:``before''});
103
104
```

```
105 CALL apoc.trigger.add(‘‘When dummy_8 happens’’,
106  ‘UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,‘‘executed’’) as prop
107   WITH prop.node as n WHERE n.reda_id="dummy_8" AND n.executed>0
108
109   MATCH (c_2)-[:response]->(n)
110   WHERE c_2.reda_id="c_2"
111
112   SET n.value = c_2.value
113
114   RETURN 1 as X
115 ’,{phase:‘‘before’’});
```

ANNEX II

Veterinarian Clinic with ReDa

II.1 ReDa Process

```
(createAppointment:DataIN) [?: {clientID:String, vetID:String}]
;
createAppointment -->> {
  (appointment:DataOUT) [{clientID:@trigger.value.clientID, vetID:@trigger.value.vetID }],
  (choosePet:DataIN) [?:{name:String, age:Number, breed:String}]
  (checkIn:DataIN) [?]
  %(checkOut:DataIN) [?]
  ;
  choosePet -->> {
    (pet:DataOUT) [{name:@trigger.value.name, age:@trigger.value.age, breed:@trigger.value.
        breed}],
    (form:DataIN) [?:{description:String, petName:@trigger.value.name}]
    ;
    ;
    pet -[:has_appointment]-> appointment
    },
  choosePet -->* checkIn,
  checkIn -->% checkIn,
  checkIn -->% choosePet,
  checkIn -->+ checkOut,
  checkOut -->% checkOut
  ;
}
;
```

II.2 Cypher Script

```
CREATE (createAppointment_0 : DataIN { reda_id: "createAppointment_0", executed: 0,
    included: true, pending: false, type: "{clientID: String, vetID: String}",
    value_clientID: "", value_vetID: ""})

CREATE (dummy_8: DUMMY {reda_id: "dummy_8", exp: "true", value: true, executed: 0,
    included: true, pending: false})

CREATE (dummy_8)-[:spawnCondition]->(createAppointment_0)

;

CALL apoc.trigger.add('EVERYWERE',
     'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties, "executed") as prop

  WITH prop.node as n WHERE n.executed>0

  CALL apoc.util.validate( EXISTS((n)<-[:condition]-({included:true, executed:0})),
                           "EVENT HAS A CONDITION UNSATISFIED", [])
  CALL apoc.util.validate( EXISTS((n)<-[:milestone]-({included:true, pending:true})),
                           "EVENT HAS A MILESTONE UNSATISFIED", [])
  CALL apoc.util.validate( n.included=false, "EVENT IS NOT INCLUDED", [])

  CALL apoc.util.validate( EXISTS((n) <-[:conditionRight]- ({value:true}) <-[:
      conditionLeft]- ({included:true, executed:0})),
                       "EVENT HAS A CONDITION EXPRESSION UNSATISFIED", [])

  CALL apoc.util.validate( EXISTS((n) <-[:milestoneRight]- ({value:true}) <-[:
      milestoneLeft]- ({included:true, pending:true})),
                       "EVENT HAS A MILESTONE EXPRESSION UNSATISFIED", [])

  RETURN 1 as X


  UNION
  UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
  WITH prop.node as n WHERE n.executed>0

  SET n.pending = false

  WITH n
  OPTIONAL MATCH (n)-[:response]->(t)
  SET t.pending = true
  WITH n
  OPTIONAL MATCH (n)-[:excludes]->(t)
  SET t.included = false
  WITH n
  OPTIONAL MATCH (n)-[:includes]->(t)
  SET t.included = true
```

```
44    WITH n
45    OPTIONAL MATCH (n)-[:responseLeft]->({value:true})-[:responseRight]->(t)
46    SET t.pending = true
47    WITH n
48    OPTIONAL MATCH (n)-[:excludesLeft]->({value:true})-[:excludesRight]->(t)
49    SET t.included = false
50    WITH n
51    OPTIONAL MATCH (n)-[:includesLeft]->({value:true})-[:includesRight]->(t)
52    SET t.included = true
53
54    RETURN 1 as X
55  ',
56  {phase:'before'});
57
58
59
60  CALL apoc.trigger.add('Trigger_id_9',
61     'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
62
63    WITH prop.node as n WHERE n.reda_id="choosePet_2" AND n.executed>0
64
65    MATCH (appointment_1)-[:appointment]->(n)
66
67    MATCH (dummy_7)-[:spawnCondition]->(n)
68    WHERE dummy_7.value = true
69
70
71    CREATE (pet_5: DataOUT {reda_id: "pet_5", executed: 0, included: true, pending: false,
          expr: "{breed: @trigger.value.breed, age: @trigger.value.age, name: @trigger.value.
          name}", value_breed :n.value_breed, value_age: n.value_age, value_name: n.value_name
          })
72
73    CREATE (form_6 : DataIN {reda_id: "form_6", executed: 0, included: true, pending: false,
           type:"{description: String, petName: String}", value_description: "", value_petName
          : ""})
74
75    CREATE (pet_5)-[:hasAppointment]->(appointment_1)
76
77
78    RETURN 1 as X
79  ',
80  {phase:'before'});
81
82
83
84  CALL apoc.trigger.add('Trigger_id_10',
85    'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
86
87    WITH prop.node as n WHERE n.reda_id="createAppointment_0" AND n.executed>0
88
```

```
89    MATCH (dummy_8)-[:spawnCondition]->(n)
90    WHERE dummy_8.value = true
91
92
93    CREATE (appointment_1 : DataOUT{ reda_id: "appointment_1", executed: 0, included: true,
          pending: false, expr: "{vetID: @trigger.value.vetID, clientID: @trigger.value.
          clientID}", value_vetID: n.value_vetID, value_clientID: n.value_clientID})
94
95    CREATE (choosePet_2 : DataIN{ reda_id: "choosePet_2", executed: 0, included: true,
          pending: false, type: "{name: String, age: Number, breed: String}", value_name: "",
          value_age: 0, value_breed: ""})
96
97    CREATE (checkIn_3 : DataIN{ reda_id: "checkIn_3", executed: 0, included: true, pending:
          false, type: "Unit"})
98
99    CREATE (checkOut_4 : DataIN{reda_id:"checkOut_4", executed:0, included:false, pending:
          false, type:"Unit"})
100
101   CREATE (dummy_7:DUMMY{reda_id:"dummy_7", exp:"true", value:true, executed:0, included:
          true, pending:false})
102
103   CREATE (appointment_1)-[:appointment]->(choosePet_2)
104
105   CREATE (checkIn_3)-[:excludes]->(checkIn_3)
106
107   CREATE (checkIn_3)-[:excludes]->(choosePet_2)
108
109   CREATE (checkIn_3)-[:includes]->(checkOut_4)
110
111   CREATE (checkOut_4)-[:excludes]->(checkOut_4)
112
113   CREATE (choosePet_2)-[:condition]->(checkIn_3)
114
115   CREATE (dummy_7)-[:spawnCondition]->(choosePet_2)
116
117
118   RETURN 1 as X
119 ',
120 {phase:'before'});
```

ANNEX III

Library ReDa Process

III.1 ReDa Process

```
(createAuthor:DataIN) [?: {authorName:String}]
;
createAuthor -->> {
  (author:DataOUT) [{name:@trigger.value.authorName}],
  (createBook:DataIN) [?:{bookTitle:String, genre:String, isbn:String}]
  ;
  createBook -->> {
    (book:DataOUT) [{bookTitle:@trigger.value.bookTitle, genre:@trigger.value.genre, isbn:
        @trigger.value.isbn, author:author.value.name}],
    (loanBook:DataIN) [?:{username:String}]
    ;
    loanBook -->> {
      (loan:DataOUT) [{user:@trigger.value.username}],
      !(returnBook:DataIN) [?]
      ;
      returnBook -->% returnBook,
      returnBook -->+ loanBook
      ;
      loan -[:BOOK]-> book,
      returnBook -[:LOAN]-> loan
    },
    loanBook -->% loanBook
    ;
    author -[:WROTE]-> book
    }
  ;
}
;
```

III.2 Cypher Script

```
CREATE (createAuthor_48:DataIN{reda_id:"createAuthor_48", executed:0, included:true,
    pending:false, type:"{authorName:String}", value_authorName:""})


CREATE (dummy_57:DUMMY{reda_id:"dummy_57", exp:"true", value:true, executed:0, included:
    true, pending:false})


CREATE (dummy_57)-[:spawnCondition]->(createAuthor_48)


;

CALL apoc.trigger.add('EVERYWERE',
                    'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed
                        ") as prop
  WITH prop.node as n
  WHERE n.executed>0

  CALL apoc.util.validate( EXISTS((n)<-[:condition]-({included:true, executed:0})),
                        "EVENT HAS A CONDITION UNSATISFIED", [])
  CALL apoc.util.validate( EXISTS((n)<-[:milestone]-({included:true, pending:true})),
                        "EVENT HAS A MILESTONE UNSATISFIED", [])
  CALL apoc.util.validate( n.included=false, "EVENT IS NOT INCLUDED", [])

  CALL apoc.util.validate( EXISTS((n)<-[:conditionRight]-({value:true})<-[:conditionLeft
      ]-({included:true, executed:0})),
                        "EVENT HAS A CONDITION EXPRESSION UNSATISFIED", [])

  CALL apoc.util.validate( EXISTS((n)<-[:milestoneRight]-({value:true})<-[:milestoneLeft
      ]-({included:true, pending:true})),
                        "EVENT HAS A MILESTONE EXPRESSION UNSATISFIED", [])

  RETURN 1 as X


  UNION
  UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed") as prop
  WITH prop.node as n WHERE n.executed>0

  SET n.pending = false

  WITH n
  OPTIONAL MATCH (n)-[:response]->(t)
  SET t.pending = true
  WITH n
  OPTIONAL MATCH (n)-[:excludes]->(t)
  SET t.included = false
```

```
44    WITH n
45    OPTIONAL MATCH (n)-[:includes]->(t)
46    SET t.included = true
47    WITH n
48    OPTIONAL MATCH (n)-[:responseLeft]->({value:true})-[:responseRight]->(t)
49    SET t.pending = true
50    WITH n
51    OPTIONAL MATCH (n)-[:excludesLeft]->({value:true})-[:excludesRight]->(t)
52    SET t.included = false
53    WITH n
54    OPTIONAL MATCH (n)-[:includesLeft]->({value:true})-[:includesRight]->(t)
55    SET t.included = true
56
57    RETURN 1 as X
58  ',
59  {phase:'before'});
60
61
62
63
64
65  CALL apoc.trigger.add('Trigger_id_58',
66                        'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed
                              ") as prop
67    WITH prop.node as n
68    WHERE n.reda_id="loanBook_52" AND n.executed>0
69
70    MATCH (book_51)-[:book]->(n)
71
72    MATCH (dummy_55)-[:spawnCondition]->(n)
73    WHERE dummy_55.value = true
74
75
76    MATCH (loanBook_52)-[:loanBook]->(n)
77
78    CREATE (loan_53:DataOUT{reda_id:"loan_53", executed:0, included:true, pending:false,
          expr:"{user:@trigger.value.username}", value_user:n.value_username})
79
80    CREATE (returnBook_54:DataIN{reda_id:"returnBook_54", executed:0, included:true, pending
          :true, type:"Unit"})
81
82    CREATE (loan_53)-[:BOOK]->(book_51)
83
84    CREATE (returnBook_54)-[:LOAN]->(loan_53)
85
86    CREATE (returnBook_54)-[:excludes]->(returnBook_54)
87
88    CREATE (returnBook_54)-[:includes]->(loanBook_52)
89
90
```

```
  RETURN 1 as X
',
{phase:'before'});



CALL apoc.trigger.add('Trigger_id_59',
                    'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed
                        ") as prop
  WITH prop.node as n
  WHERE n.reda_id="book_51" AND n.executed>0

  MATCH (author_49)-[:author]->(n)

  SET n.value_author=author_49.value_name



  RETURN 1 as X
',
{phase:'before'});




CALL apoc.trigger.add('Trigger_id_60',
                    'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed
                        ") as prop
  WITH prop.node as n
  WHERE n.reda_id="createBook_50" AND n.executed>0

  MATCH (author_49)-[:author]->(n)

  MATCH (dummy_56)-[:spawnCondition]->(n)
  WHERE dummy_56.value = true


  CREATE (book_51:DataOUT{reda_id:"book_51", executed:0, included:true, pending:false,
      expr:"{isbn:@trigger.value.isbn, genre:@trigger.value.genre, bookTitle:@trigger.
      value.bookTitle}", value_isbn:n.value_isbn, value_genre:n.value_genre,
      value_bookTitle:n.value_bookTitle})

  CREATE (loanBook_52:DataIN{reda_id:"loanBook_52", executed:0, included:true, pending:
      false, type:"{username:String}", value_username:""})

  CREATE (dummy_55:DUMMY{reda_id:"dummy_55", exp:"true", value:true, executed:0, included:
      true, pending:false})

  CREATE (author_49)-[:WROTE]->(book_51)
```

```
134
135   CREATE (author_49)-[:author]->(book_51)
136
137   CREATE (author_49)-[:condition]->(book_51)
138
139   CREATE (author_49)-[:response]->(book_51)
140
141   CREATE (book_51)-[:book]->(loanBook_52)
142
143   CREATE (dummy_55)-[:spawnCondition]->(loanBook_52)
144
145   CREATE (loanBook_52)-[:excludes]->(loanBook_52)
146
147   CREATE (loanBook_52)-[:loanBook]->(loanBook_52)
148
149
150   RETURN 1 as X
151 ',
152 {phase:'before'});
153
154
155
156
157
158
159
160 CALL apoc.trigger.add('Trigger_id_61',
161                       'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed
                              ") as prop
162   WITH prop.node as n
163   WHERE n.reda_id="createAuthor_48" AND n.executed>0
164
165   MATCH (dummy_57)-[:spawnCondition]->(n)
166   WHERE dummy_57.value = true
167
168
169   CREATE (author_49:DataOUT{reda_id:"author_49", executed:0, included:true, pending:false,
           expr:"{name:@trigger.value.authorName}", value_name:n.value_authorName})
170
171   CREATE (createBook_50:DataIN{reda_id:"createBook_50", executed:0, included:true, pending
          :false, type:"{bookTitle:String, genre:String, isbn:String}", value_bookTitle:"",
          value_genre:"", value_isbn:""})
172
173   CREATE (dummy_56:DUMMY{reda_id:"dummy_56", exp:"true", value:true, executed:0, included:
          true, pending:false})
174
175   CREATE (author_49)-[:author]->(createBook_50)
176
177   CREATE (dummy_56)-[:spawnCondition]->(createBook_50)
178
```

```
179
180   RETURN 1 as X
181 ',
182 {phase:'before'});
```