André Gonçalves Fragoso

Degree in Computer Science and Engineering

Deducing Operation Commutativity from Replicated Data Declaration

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

Master of Science in
Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Francisco de Moura e Castro Ascensão de Azevedo
Raporteur:	Francisco Cipriano da Cunha Martins
Member:	Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

January, 2020

Deducing Operation Commutativity from Replicated Data Declaration

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

To my parents.

Acknowledgements

I would like to start by thanking my adviser, Hervé Paulino, for his availability for constant meetings throughout the development of this thesis and also for his guidance and valuable support during the whole process. Evidently, I also have to thank António Ravara and Marco Giunti for their relevant contributions and for the opportunity to collaborate and learn from them. I am truly grateful to have been part of this project and to have been able to work in such a promising topic.

Moreover, I would like to thank the Department of Informatics of the NOVA University of Lisbon for providing me all the resources and working conditions needed along the course and all its members who contributed to my academic and personal development. Additionally, I would also like to give a special thanks to the Faculty of Sciences and Technology of the NOVA University of Lisbon for supporting my attendance at the INForum 2019 conference which allowed me to gather some reviews and suggestions regarding the work described in these pages.

This work was partially supported by the Fundação para a Ciência e Tecnologia (FCT) in the context of the DeDuCe project (PTDC/CCI-COM/32166/2017) and the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) research centre (UID/CEC/04516/2013), which I would like to thank for their financial support.

And last but not least, I also want to express my gratitude to my closest friends and family who have always supported me throughout my academic journey although they did not see me very often and, particularly, for their encouragement to get my thesis done without even understanding its subject despite the countless times I have tried to explain.

To everyone who contributed directly or indirectly to the accomplishment of this thesis, thank you!

Abstract

Distributed systems often resort to data replication not only to enhance their availability but also to reduce user-perceived latency by balancing the load between replicas and routing their requests accordingly. The choice of which consistency level that should be adopted by these replicated systems is critical for the fulfilment of their performance and correctness requirements. However, defining a strategy that strikes the right balance between these concerns in this type of environments is far from being a trivial task due to the related overheads that are amplified in distributed scenarios.

Recognising the tension between latency and consistency, many systems allow multiple consistency levels to coexist. Nevertheless, the performance fine-tuning mechanisms supported by the existing hybrid solutions place a high burden on the programmer since the necessary input can be somehow complex requiring him to understand the semantics of each operation of the service he is developing in order to correctly instruct the system on how to handle concurrent updates. Thus, specifying operation dependencies, orderings and invariants to be preserved or even picking the right consistency level to be assigned to a certain data item is, generally, an error-prone task that hinders reasoning.

To overcome this adversity, this work aims to reduce the effort spent by the programmer by only requiring the latter to introduce a simple and intuitive input at data declaration. Following this approach, reasoning is centralised and all accesses to replicated data are identified automatically. With all data accesses identified, it is then possible to deduce the side effects of each operation and determine, for each one of them, those with which it conflicts. In this context, this thesis also presents a compile-time analysis applied to the Java language able to evaluate operation pairwise commutativity from the input given at data declaration.

Keywords: replicated systems, consistency levels, data-centric concurrency control, language interpretation

Resumo

Os sistemas distribuídos recorrem frequentemente à replicação de dados não só para reforçar a sua disponibilidade, mas também para reduzir a latência observada pelos seus utilizadores através da distribuição de carga entre réplicas. A escolha do nível de consistência a ser adotado por estes sistemas replicados é crucial para o cumprimento dos respetivos requisitos de desempenho e correção. Contudo, definir uma estratégia que alcance um equilíbrio apropriado relativamente a estes conceitos neste tipo de ambientes está longe de ser uma tarefa trivial devido às sobrecargas acrescidas introduzidas pelos cenários distribuídos.

Atendendo às incompatibilidades entre propriedades como a latência e a consistência, diversos sistemas permitem que vários níveis de consistência coexistam entre si. No entanto, ao regular os níveis de desempenho destas soluções híbridas o ónus recai sobre o programador, já que para instruir corretamente o sistema para lidar com operações simultâneas este poderá ser obrigado a entender a semântica de cada operação do serviço. Assim, ordenar operações, especificar dependências e invariantes a serem preservadas ou, até mesmo, escolher o nível de consistência correto a ser atribuído a um determinado elemento dos dados são, geralmente, tarefas propensas a erros cujo raciocínio é dificultado.

De forma a superar esta adversidade, este trabalho visa reduzir o esforço do programador exigindo apenas uma contribuição simples e intuitiva por parte do mesmo ao nível da declaração dos dados. Seguindo esta abordagem, o raciocínio é centralizado e todos os acessos aos dados replicados são identificados automaticamente. Com todos estes acessos identificados, torna-se possível deduzir os efeitos das operações e determinar aquelas com as quais cada operação entra em conflito. Neste contexto, esta tese apresenta também uma análise em tempo de compilação aplicada à linguagem Java capaz de avaliar a comutatividade entre pares de operações a partir das anotações especificadas ao nível da declaração dos dados.

Palavras-chave: sistemas replicados, níveis de consistência, controlo de concorrência centrado nos dados, interpretação de linguagens

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

We live in a fast-paced world where the performance and the quality of the user experience provided by Internet services play an important role on the general acceptance of their consumers [32]. Therefore, it is intolerable that these services become unavailable due to failures or that their clients have to wait a large amount of time until they receive a response. Taking into account these aspects, distributed systems often rely on data replication. Data is replicated not only to enhance availability but also to reinforce data durability through redundancy and to reduce user-perceived latency by balancing the load between replicas. Despite being very useful and suitable, replicated systems are not a panacea in the sense that one still has to reason about the trade-offs between the above mentioned concerns in order to make sure that all replicas behave in such a way that the system invariants are preserved and the user experience is not compromised.

The underlying infrastructures of many large-scale systems also resort to geo-replication to further improve user experience [7, 8, 33]. Consequently, these systems can easily scale their online services by distributing their data on data centres scattered across the globe, being able to minimise latency by placing replicas closer to their clients and by routing requests accordingly. However, it is even more demanding to follow an approach that seeks to strike the right balance between availability, consistency and access time in this type of environments due to the corresponding overheads introduced by these geographically distributed scenarios.

According to the CAP theorem [6, 12] it is not possible for a distributed system to simultaneously be (strongly) consistent[1], (highly) available and partition tolerant. Thus,

[1] The consistency property presented in the CAP theorem is completely different from the one defined in the ACID properties where no transaction can create an invalid data state.

these systems can only support at most two of the following three properties at a given point in time:

- **(C)onsistency** - every read operation returns the most recent written value;
- **(A)vailability** - all read/write requests will eventually succeed;
- **(P)artition-tolerance** - the system continues to operate in spite of network partitions where some nodes may be unavailable.

A relevant observation is that in large-scale systems, network partitions need to be tolerated. Therefore, these systems cannot ensure strong consistency semantics and high availability at the same time. Some modern systems, like Cassandra [20] and Amazon DynamoDB [9], have chosen to support availability and partition tolerance at the expense of strong consistency which allows these systems to provide low latency for client operations and achieve high scalability. Nonetheless, the choice of the two properties to consider obviously depends on the type of service to be provided. For instance, a banking service (which is one of the most commonly used example in the literature and which we will adopt throughout this document to clarify certain subtleties) always needs its data to be consistent and, therefore, its availability is neglected as opposed to the other two properties (tolerance to network partitions and stronger consistency levels).

1.2 Motivation

On the one hand, to cope with the demand for fast response times, one might end up wanting their clients to communicate with the lowest possible number of replicas to process their operations. On the other hand, to ensure that all replicas evolve according to the same states, one might end up wanting their clients to communicate with a majority of the replicas in the system instead. In order to provide short response times, systems with weaker consistency guarantees usually allow operations to be performed on a single replica, as opposed to systems with stronger consistency guarantees that require cross-site synchronisation.

Several systems resort to weaker consistency semantics to avoid the need for synchronisation between replicas. Under weak consistency levels, as the state of the replicas might temporarily diverge, correcteness can be violated by way of concurrent updates. Contrarily, some systems resort to stronger consistency semantics in order to reach global synchronisation even though it may jeopardise the performance and scalability of their services. Hence, picking the right level of consistency requires a thorough prior study on the pros and cons of all possible approaches since the chosen consistency semantics have a huge impact on the behaviour of the system.

The aforementioned dilemma reproduces one of the biggest challenges that arises when setting up these replicated data systems that are at the core of current Internet

services and addresses the tension between low latency and strongly consistent responses in this type of environments. Recognising this tension, some solutions attempted to circumvent the limitations imposed by the chosen consistency semantics with respect to the system performance by allowing multiple levels to coexist, i.e., by letting different operations execute under different consistency levels [19, 35, 37]. Consequently, due to the fact that they are not confined to a single consistency level, these systems are able to provide a faster service by weakening (when possible) the consistency level under which their operations are performed. However, these hybrid solutions place a high burden on the developer since the latter has to decide which consistency level should be assigned to each operation taking into account the system invariants that must be satisfied. For that reason, the developer needs to understand the semantics of each operation and reason about their behaviour when running at different consistency levels. That being said, there is still one challenge currently open: how to deduce which is the consistency level required by a given operation and how to perform such deduction automatically with small and intuitive input from the programmer.

1.3 Problem Statement

To achieve an automatic deduction of the consistency level each operation requires, it is mandatory that the programmer instructs the system on how it should handle concurrent requests or, in other words, how concurrency control mechanisms should act so that the system does not reach undesired states. Concurrency control mechanisms are the coordination techniques responsible for preventing incorrect outcomes yielded by concurrent executions. This prevention is driven by information that can be provided at the operation level or directly at data declaration level. Hence, one can employ a control-centric concurrency control or a data-centric concurrency control.

Control-centric approaches require the programmers to annotate the operations of their service with rules that prevent incorrect outcomes and help the system managing concurrent requests. Such solutions, like the ones presented in [36] and [3], are error-prone and their reasoning is decentralised since that even the smallest mistake may lead to the initial specification not being met, arising the need to keep track of all data accesses to ensure that this does not occur. Alternatively, data-centric approaches shift all user-given information to data declaration. These approaches have two advantages: they centralise reasoning, making it easier to reason as code scales, and they automatically keep track of all accesses to replicated data, removing the burden from the programmer. Although there are some proposals that apply data-centric reasoning to data replication, they either force the same data item to be always accessed with the same consistency level [41] which is not desirable in many scenarios or are confined to a specific type of systems and limited by the operations that are provided by the libraries they rely on [24], and thus, cannot be considered as generic approaches.

1.4 Contributions

This work presents two main contributions:

- The extension of data-centric concurrency control to cope with replicated data systems by means of a compile-time analysis applied to the Java language able to evaluate operation pairwise commutativity from the input given at data declaration.
- An experimental evaluation of the proposed solution that shows its correctness and validates its usefulness when implementing replicated services.

It is important to take into account that, as this is the first work that follows the approach we came up with, the commutativity analysis presented throughout this document only covers operations over primitive data types and private replicated fields.

1.4.1 Publications

The work presented in this document generated the following publication:

- **Identificação de Comutatividade de Operações em Sistemas Replicados** André Fragoso, Hervé Paulino, Marco Giunti and António Ravara. Proceedings of the 11th Simpósio de Informática (INFORUM'19), Guimarães, Portugal, September 2019

1.5 Document Structure

The remainder of this document is organised as follows:

- **Chapter** 2 introduces the fundamental background and related work. Some consistency models as well as their corresponding properties are detailed and the most relevant existing systems that take advantage of each one of them are described. We also explain how several consistency semantics can coexist with each other based on two different models, we consider the main drawbacks of the existing concurrency control approaches which our solution aims to overcome and we, additionally, mention some relevant concepts regarding operation commutativity.
- **Chapter** 3 details the shared memory model for concurrency control that was adapted to address replicated systems and specifies the corresponding modifications that were introduced to it. Our solution is briefly described and we also show how the information our compile-time analysis produces can be useful at runtime.
- **Chapter** 4 describes the implementation details of each phase of our solution as well as the options that were made along the development process to broaden the space of non-conflicting operations (i.e., the scope of potentially commutative operations).

- **Chapter 5** presents the evaluation methodology used to analyse the feasibility of adopting our solution when setting up a given replicated service. The results obtained in this evaluation step, which was based on two metrics (correctness and performance), are discussed and analysed against different scenarios in order to understand the impact that certain parameters have on the performance of the developed solution.
- **Chapter 6** summarises the conclusions that were taken from the elaboration of this thesis and provides a number of ways in which our solution can be extended.

Chapter 2

Related Work

This chapter covers the main concepts that one needs to be familiar with in order to be able to understand the focus of the work that was developed and the gaps it aims to fulfil. Firstly, all the consistency models that one needs to know in order to comprehend the solutions that take advantage of them are explained and examples of well-known systems that implement each one of them are briefly described. Then, two models that allow the coexistence of multiple consistency levels are specified to better understand how these operate. And, finally, it is presented a detailed overview of the existing concurrency control approaches (control-centric and data-centric) and their corresponding limitations are discussed.

2.1 Consistency Levels

Many consistency models have been proposed in the literature but not all of them are capable of offering fast and consistent responses in replicated systems. The adopted consistency model delimitates the anomalies that the system exhibits, that is, it restricts the states that a client can observe. Thus, the choice of one of the following consistency semantics depends on the application requirements.

The consistency levels mentioned below are divided into non-transactional and transactional semantics and, for ease of understanding, these models are accompanied by examples that demonstrate their outcomes under specific circumstances.

2.1.1 Non-Transactional Consistency Semantics

Non-transactional consistency models establish the set of possible results from concurrent individual operations requested by the clients. The models that fit within this scope (i.e., the consistency levels that do not consider sequences of operations) are described

next and are presented from the weakest models (where clients eventually receive a response) to the strongest ones (under which systems could not be totally available).

Eventual consistency [31, 39] is on the weakest side of the consistency spectrum. Under this model, operations are executed in a small set of replicas (possibly in a single replica). The only guarantee is that, when write operations cease, all replicas will eventually converge to the same state. This model enables updates to be asynchronously propagated to all replicas asynchronously after clients obtain their responses, which implies that the state between them can differ provisionally and thus clients can observe stale data. There are multiple alternatives to achieve state convergence, such as the *last-writer-wins* policy.

There is a particular case of eventual consistency called strong eventual consistency (SEC) [34] that is only valid for certain data types. Conflict-free Replicated Data Types (CRDTs) [29, 34] are a common approach to ensure this type of consistency. As these data types encapsulate their own merge semantics, they know how to incorporate concurrent updates in a deterministic way so that all replicas eventually converge to the same state. Therefore, contrarily to eventual consistency, there is no need for conflict resolution.

DNS (Domain Name System) [26] is the most popular service that provides eventual consistency. It is a hierarchical name service where write operations are managed by a single replica (master) and the corresponding updates are then propagated to the remaining replicas (slaves). Since read operations can be directed to any node, if a client issues a read operation to a replica that has not been yet notified, old values can be observed.

Timeline consistency [27] provides a total ordering on all updates (even from different clients) to the same data item according to a certain timeline shared by all replicas. If a client needs to access a different replica, a read operation may return a consistent but dated view of the system state (moving back in the timeline with respect to previous read operations).

Yahoo!'s PNUTS [27] is a globally-distributed database system that provides per-record timeline consistency where all replicas of a given record apply all updates in the same order by using a per-record master that is responsible for executing all updates on a given record. Its API allows the specification of the consistency level of read operations to be specified: one can request the most up-to-date copy of a certain record (latest version that reflects all writes) or allow for potentially stale versions.

Causal consistency [1] is one of the strongest weak consistency models. Under causal consistency a distinction is made between causally related operations and concurrent operations. If one operation influences another then they are causally related, otherwise, they are concurrent operations. Since the data returned by read operations have to respect the happened-before partial order [21], all replicas must agree on the order that causally

dependent write operations are applied, while concurrent writes can occur in any order and can be applied in different orders by different replicas. Causal consistency is typically captured by the aggregation of the following properties:

- *Read your writes* - read operations issued by a given cient will always reflect the effects of all previous write operations issued by him. Clients will always witness a state that is at least as up-to-date as the state written by themselves;
- *Monotonic reads* - subsequent reads issued by the same client should observe either the same state or a state modified by new write operations;
- *Monotonic writes* - all write operations requested by a given client will become visible respecting the order by which they were issued;
- *Writes follow reads* - if a client observes the effects of a write operation in its session, subsequent write operations issued by that client must be ordered after the previously observed write operation.

Contrarily to eventual consistency, causality by itself does not guarantee state convergence even if at some point in time no more write operations occur. Causal consistency and timeline consistency are not really comparable since causal consistency focuses on clients and the states they can observe and timeline consistency focuses on data and the sequence of operations performed on each item at all replicas instead.

Lazy replication [19] is a technique that supports causal ordering of operations. This system provides causal consistency by assigning timestamps to data items (which can be seen as its version) when replicas are requested to update them. To be able to infer whether a given operation can be performed or not at a certain replica, the system relies on this information to determine if all operations on which the requested operation depends on have already been applied at such replica.

Causal+ Consistency [25] is a consistency model that seeks to provide both eventual and causal consistency guarantees. Summarily, it preserves the ordering of causally-related operations on all replicas and ensures state convergence when no write operations occur.

COPS (Clusters of Order-Preserving Servers) [25] is a key-value storage system that provides causal+ consistency. It takes advantage of dependency metadata that is included in both read and write requests to capture causality. Before exposing updates, the system checks whether causal dependencies are satisfied in the local data centre and, if not, operations are delayed until the needed version is written. Its default version uses the *last-writer-wins* policy to handle conflicting updates which ensures that replicas never permanently diverge and that conflicting updates to the same data item are identically

handled at all sites.

Sequential Consistency [22] implies that all operations in a system are executed in some total order that respects the order by which they were requested by each client. Read operations may return stale values depending on the generated sequence but they never go back in time that is, after observing a certain data version, the client will never observe previous ones. Despite not being the strictest consistency model, it requires global synchronisation which hurts performance.

ZooKeeper [16] is a coordination service for distributed applications that provides sequential consistency. The node that the client is connected to forwards all write operations to an elected leader which ensures that the writes will be sequential by ordering all updates with timestamps that reflect their total order and by implementing a consensus protocol where a majority (quorum) of nodes have to acknowledge an update for it to be considered successful which ensures consistent responses. Conversely, read operations are executed at the node that the client is connected to so they may return old values.

Linearisability [14] is on the strongest side of the consistency spectrum where replicated systems behave as if a single replica handles all operations (single storage illusion), making them easier for a programmer to reason about. This, however, requires coordination among replicas to agree on the order in which operations are performed which leads to poor performance and hinders scalability. Under this consistency model, once a write operation completes, all subsequent read operations must return the value written by that operation or by subsequent ones. Once a read operation returns a value, subsequent read operations cannot return a value related to an older system state.

Chain replication [38] is an approach that grants linearisability. As the name implies it assumes that replicas are arranged in a chain topology (linked nodes). While write operations are directed to the head of the chain and are then propagated until they reach its tail, read operations are directed to the tail of the chain where the corresponding replica already reflects the requested updates. If one of the replicas fails, the throughput levels drop until the chain is repaired.

2.1.1.1 Comparison

Understanding what differentiates one weak consistency model from another is straightforward considering the scenario presented in Figure 2.1. Under this scenario whose outcomes are highly influenced by the consistency model under which it is executed, one can realise what distinguishes the presented weak consistency semantics (eventual, causal and timeline consistency). For ease of understanding we will assume that the read operation requested by the client c_2 returns the value written by c_1 and that the initial

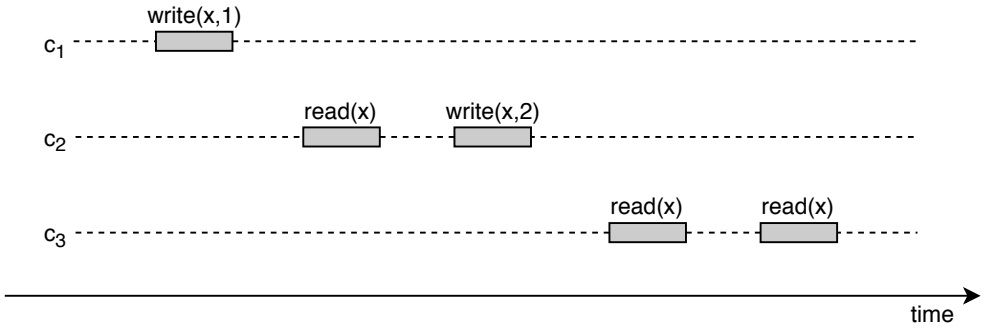


Figure 2.1: Scenario where the dissimilarities between weak non-transactional consistency models are easily spotted.

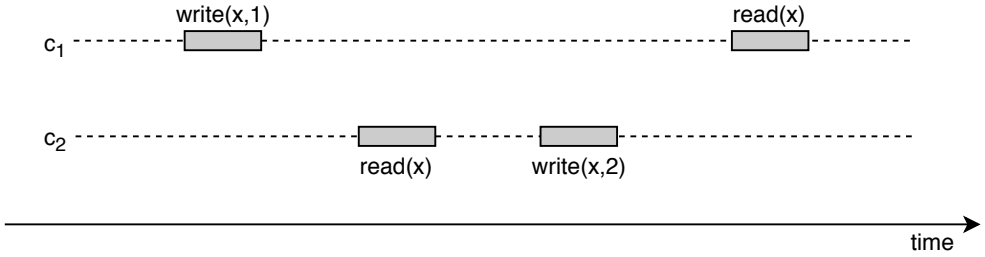


Figure 2.2: Scenario where the dissimilarities between strong non-transactional consistency models are easily spotted.

value of x is 0. If we execute the given scenario under causal consistency both write operations from c_1 and c_2 are causally related which imposes an order on the states that can be observed by the clients. Therefore, regardless of the replica that c_3 communicates with, both its read operations must follow the previous imposed order: if the first read operation returns 1, the following read must return 1 or 2; if the first read operation already returns 2, the following read cannot return neither 0 nor 1; if the first read operation does not reflect any of the previous updates (returns x's initial value), the following read can return 0, 1 or 2. Under timeline consistency, the states that clients can observe must follow the order (the timeline) it imposes on write operations. Thus, c_3 may observe 2 and then 1 as a result of its read operations if c_1's write operation is ordered after c_2's or if c_3 has to request its second read operation to a different replica and the latter is out of sync. Finally, under eventual consistency any order of the observed states is acceptable since this model does not provide any guarantees.

Given the depicted scenario in Figure 2.2, one can easily realise what distinguishes the presented strong consistency semantics (linearisability and sequencial consistency). When executing those operations under linearisability, both read operations requested by c_1 and c_2 will certainly return values 2 and 1, respectively. However, under sequential consistency the system may interleave operations while maintaining the order by which operations were requested. Consequently, both read operations may not provide the most recent value that was written (e.g., the read operation requested by c_1 will always reflect the write operation previously requested by that client but it is not certain that it will

reflect the write operation issued by c_2).

2.1.2 Transactional Consistency Semantics

Transactional consistency models, on the other hand, refer to sequences of operations that must be evaluated as a unit (transactions). Either all operations take effect (committed transaction) or none does (aborted transaction). These consistency models only consider committed transactions and their strength is measured by the isolation level they offer, that is, the concurrency effects that clients may encounter. The models that suit this context of transactional systems are described next and are presented from the model that offers the highest isolation level to the one that provides the lowest isolation level.

Serialisability [5, 27] is the highest level of isolation between transactions. Under this model, the concurrent execution of a set of transactions (possibly over different data items) is guaranteed to be equivalent to some serial schedule (total ordering without transaction overlap) of those same transactions. Thus, it is required that the concurrently requested transactions must be executed in a single order across all replicas.

Google Spanner [8] is a globally distributed database used internally by Google for several of its own services. It not only supports serialisability where all transactions appear as if they executed in a serial order (even if some of them actually occurred in parallel), but it also guarantees that this serial order corresponds to real time (stronger guarantee). To achieve this level of consistency, it assigns global commit timestamps to all transactions relying on a highly reliable wall-clock time tracking service that is built on GPS and atomic clock hardware.

Snapshot Isolation (SI) [4] is a relaxation of serializability. Under SI, there is no guarantee that the state that is observed by a certain transaction reflects the most recent updates. It only guarantees that transactions will witness a consistent snapshot (the last committed values by the time it started) and concurrent updates will not be visible. The transaction will not succeed if it attempts to update data that has changed since it began.

YugaByte DB is a globally distributed and scalable transactional database that provides SI [40]. In order to support this isolation level, this system takes advantage of write locks that, as the name implies, lock the data items that the transaction attempts to modify (naturally, this type of locks conflict with each other and the set of data items written by concurrent transactions must be disjoint).

Parallel Snapshot Isolation (PSI) [37] is a transactional consistency model that relaxes SI. It provides better scalability and availability in geo-replicated scenarios by handling

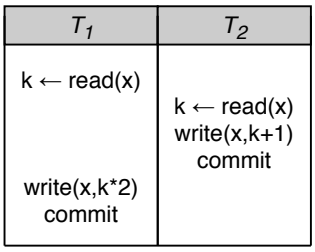


Figure 2.3: Schedule where the subtle differences between transactional consistency models are easily demonstrated (serialisability and snapshot isolation).

concurrent transactions performed at the same replica and at multiple replicas in a different manner. If two concurrent transactions do not conflict with each other (even if they are executed at different replicas) they can both commit their changes and the system can inform other replicas about their updates. If two conflicting transactions are executed at the same replica, one of them has to abort. However, if they are executed at different replicas, they both may end up committing their updates at their corresponding replica and the remaining ones must follow an appropriate order (according to some criteria) to apply them. This situation can happen if the conflict is not detected in time and thus their outcome depends on how the system manages the propagation of information among replicas.

Walter [37] is a geo-replicated key-value store that supports transactions and provides PSI. Transactions are ordered according to vector timestamps that reflect the state snapshot observed by each transaction. To prevent conflicts this system applies two techniques: data items are assigned to preferred sites enabling efficient writes since the system does not have to check every replica for write conflicts and if a certain data item is subject to regular updates from different sites, commutative operations on the item are compiled and can then be executed in any order.

2.1.2.1 Comparison

The schedule shown in Figure 2.3 could not be executed under serialisability because it is a *non-serialisable schedule* (it is not equivalent to a serial schedule) - if T_2 is ordered after T_1 the final value of x is 1 (assuming an initial value of 0), otherwise, x's final value is 2. Under SI, both transactions will observe the same state ($x = 0$) and since both attempt to modify the same data item, they conflict with each other. Their outcome depends on the policy used, for instance: if a *first-commiter-wins* policy is adopted, T_2 succeeds, T_1 aborts and the final outcome is $x = 1$.

2.2 Coexistence of Multiple Consistency Levels

Many efforts have been made to either create new solutions or extend existing ones that are able to deduce the consistency semantics required by operations to ensure that the system invariants are not violated. These solutions remove the burden from the programmer since the latter does not need to explicitly determine under which consistency level each operation should be applied in a way that the correct system behaviour is not jeopardised.

RedBlue consistency is a two-level consistency model proposed in [23] and it is a solution that allows different operations to run under different consistency levels, classifying them into: blue operations, where it is only necessary to guarantee low levels of consistency being, consequently, fast; and red operations, that are slow since it is necessary to guarantee stronger consistency semantics which imply cross-site synchronisation. In other words, this model is based on eventually consistent blue operations whose order of execution may differ between sites and strongly consistent red operations that must be executed in the same order at all sites. This consistency model also preserves causality by guaranteeing that all actions that a certain operation depends on when it is performed at one replica are still applied first when it is included at the remaining ones.

Intuitively, low latency requires an abundance of blue operations since red ones are totally ordered which negatively impacts both latency and throughput. However, in order to achieve state convergence and invariant preservation, every blue operation has to be globally commutative, i.e., they must commute with all other operations (despite their ordering, the result is the same), blue or red. For instance, considering a toy banking application that stores the balance of several accounts and where *deposits*, *withdrawals* and *interest accruals* can be performed: while a *withdrawal* must be red because it can break the non-negativity invariant, both *deposit* and *interest accrual* may be blue. However, these last two operations (addition and multiplication) do not commute. Rather than classifying both as red operations, RedBlue consistency uses a technique that was considered when building our own solution. It attempts to increase commutativity broadening the space of potentially blue operations by decomposing operations themselves into two components: a *generator operation* that identifies the changes the original operation should make but has no side effects itself (executed against the state of the primary site) and a *shadow operation* (the ones that are labelled red or blue) that performs the identified changes and is propagated and applied at all sites (including the primary site). Hence, by decoupling the identification of the side effects from their application to the state, this model is able to increase operation commutativity by rearranging *shadow operations* of non-commutative operations in such a way that they become commutative (e.g., it can make *deposits* and *interest accruals* commute by computing the amount of interest first and then handle that value as a *deposit* - two additions). Experimental results presented in [23] show that with this decoupling, RedBlue consistency significantly improved the performance of geo-replicated systems.

The work in [13] not only proposes a hybrid consistency model but it also comes

up with a proof rule capable of inferring if the consistency level chosen for a certain operation is enough to ensure that integrity invariants are not violated. Although the programmer has to intervene, this proof rule is modular since it does not require the programmer to reason about the interplay of all operations. Therefore, programmers can reason about how strengthening or weakening the consistency levels of certain operations affects correctness under some assumptions on the behaviour of other operations.

Similarly to RedBlue consistency, this approach provides causal consistency by default. However, in some cases, the guarantees provided by this consistency level may be too weak to preserve certain integrity invariants. Due to this limitation, this consistency model allows the programmer to strengthen causal consistency by specifying which operations may not be executed under that consistency level (those that need synchronisation), introducing the notion of *token acquisition*. In this model, acquiring a token denies the permission of other replicas to concurrently perform changes that require conflicting tokens.

Informally, this model guarantees that operations that acquire tokens conflicting with each other have to be causally dependent and that all replicas see such operations in the same order due to causal message propagation. This allows the weakening of the commutativity requirement only for operations that do not acquire conflicting tokens (like blue operations in RedBlue consistency) which is enough to ensure state convergence between replicas.

With this hybrid approach, the programmer is able to express some of the existing consistency models, such as: causal consistency - baseline model obtained without acquiring any token; sequential consistency - obtained by requiring every operation to acquire a mutual exclusion token; and even RedBlue consistency - red operations are guaranteed sequential consistency (they acquire tokens) and blue operations only causal consistency (they do not acquire any token). To ensure that the non-negativity invariant of the previously presented banking application is not violated, the *withdrawal* operation has to acquire a token conflicting with itself so that two *withdrawals* cannot be performed simultaneously. The remaining operations do not acquire any tokens: *deposits* and *interest accruals* can be causally independent with all operations and thus replicas are able to execute these operations without coordination.

2.3 Expressing Multiple Consistency Levels

Knowing that several consistency levels can coexist, it is necessary to understand how the programmer has to instruct the system about the rules it must follow. Obviously, the simplicity of this input is crucial for the correct behaviour of the system. If it is somehow complex for the programmer to reason about it, he is more likely to make some mistakes that can lead the system invariants to be compromised. Existing solutions require these instructions to be specified either at operation level or at data declaration level.

2.3.1 Control-Centric Approaches

Explicit Consistency [3] is an alternative consistency model for geo-replicated systems that proposes the annotation of a service interface with invariants and postconditions (information about the side effects of operations) in the form of first-order logic formulas that comprise user-defined predicates to establish the consistency rules that the system must satisfy. Given these application-specific invariants and postconditions, a system that supports this type of consistency is able to detect which sets of operations may lead to an invariant violation when executed concurrently. Programmers can select two different techniques to handle these operations: violation-avoidance (to avoid their concurrent execution) or invariant-repair (to allow conflicting operations to execute concurrently and to repair invariant violations after).

The sets of concurrent operations that may lead to an invariant violation are identified by a static analysis of the postconditions of operations performed against the system invariants and the application code is instrumented with the proper calls to a middleware library that provides functions that reflect both violation-avoidance and invariant-repair mechanisms. By incorporating these calls on the operations, a safe version of the application is derived in which all specified invariants will be preserved.

This model does not focus on the execution order of the operations. Explicit Consistency is a consistency model for application correctness centred on the application semantics, being free to reorder the execution of operations at different replicas, provided that the predefined invariants are not violated.

Another existing control-centric solution is QUELEA [36]. It is a Haskell library for declarative programming over eventually consistent stores that runs on top of Cassandra. This model is equipped with an expressive contract language where the set of legal executions allowed over the replicated data are specified. QUELEA considers finer-grained consistency properties than Explicit Consistency since its contracts are constructed using primitive consistency relations such as visibility and session order, whereas the previous consistency model contemplates invariants and postconditions. For example, recalling the previous banking example: given two *withdrawal* operations on the same bank account, one of them has to be visible to the other (one has to witness the effects generated by the other); given an operation that returns the current balance, it has to reflect the effects of all previous operations performed on the same account in the same session (sequence of operations performed by the client).

QUELEA's goal is to identify the proper consistency level for each operation such that the constraints presented in the contract are not violated. For that purpose, a static contract classification procedure maps these high-level application contracts to appropriate low-level consistency guarantees provided by the underlying store. This contract classification mechanism is completely performed at compile time and incurs no runtime overhead. This approach also provides a rule that evaluates if a given contract is well-formed by checking if it is satisfiable under the strongest possible consistency

guarantee that the store can provide.

Similarly to contracts on individual operations, this solution is also able to perform an automatic classification over contracts that express consistency specifications over transactions even though Cassandra does not provide general-purpose transactions. However, despite being this flexible, this contract language is applied to the specification of fine-grained consistency properties, not application invariants. The static analysis of this model ensures that the contract is followed, but it does not ensure that the integrity invariants are not violated.

2.3.2 Data-Centric Approaches

SIEVE [24] automatises the choice of RedBlue consistency levels for systems that use replicated databases. In order to adapt existing applications to the RedBlue consistency model, SIEVE has to transform each operation into a *generator* and a *shadow operation* and identify which *shadow operations* may break some system invariant in order to label correctly. To this end, it only requires the programmer to specify the application invariants that must be satisfied and to provide annotations regarding merge semantics (more specifically, the proper CRDT type) to handle concurrent updates that can be declared on a per-table and per-attribute basis. To adapt an application to use SIEVE, one has to replace the original JDBC[1] driver by the driver provided by SIEVE so that it can map each database update to the appropriate merge semantics and swap the initial operations with the operations over the corresponding CRDT.

This tool uses program analysis to identify non-conflicting (i.e., commutative) *shadow operations* that might break system invariants when executed under weak consistency semantics and runs them under strong consistency semantics. SIEVE splits the labelling into a potentially expensive static part and an efficient check at runtime: the static analysis generates all the possible combinations of CRDT operations (templates) that include *shadow operations* and identifies for each one of them a logical condition (weakest precondition) that ensures that the previously specified invariants are satisfied. At runtime, it is necessary to evaluate the weakest precondition to classify each operation as red or blue. For instance, if the weakest precondition does hold in a given scenario, it means that any *shadow operation* associated with that template is invariant-safe and the operation should be labelled blue (or red, otherwise).

Unfortunately, despite being close to our goal, this is not a generic approach since it is confined to databases and it is limited by the CRDT library. It also requires reasoning on merge semantics (choice of the proper CRDT semantics). Moreover, if a given operation does not preserve the system invariants under weak consistency, SIEVE's analysis suggests to execute that same operation under strong consistency. However, it does not check that the result will indeed validate those same invariants.

[1] JDBC stands for Java Database Connectivity. It is a Java API to connect, issue queries and handle results from the database.

An alternative model is the DCCT (Data-Centric Cloud Types) language proposed in [41]. This model allows developers to associate objects that are related by some integrity invariant into collections of disjoint regions. For each region, the developer has to declare the consistency level that needs to be granted to ensure that the system invariants are not compromised. In this model, reasoning is done at read/write level where sequences of atomic read–write operations may interact with multiple regions and their behaviour is dictated by the consistency policy of the corresponding regions (more flexible than relational database transactions where a single isolation level is imposed for all operations it concerns). This model allows the customisation of the consistency policy of read operations. Read operations may be requested at a weaker consistency level than the declared level of the region to which that data item belongs. However, this model does not allow read operations to be requested at a consistency level stronger than the declared level of the requested region because it cannot be enforced without also changing the consistency level of write operations. On the other hand, the consistency of write operations cannot be customised (selecting stronger consistency semantics could threaten performance and selecting weaker consistency semantics could lead to integrity invariant violation).

The main drawback of this solution is that it forces data items to be always accessed with the same consistency level. Contrarily to previous models, where it was possible for operations with multiple consistency levels to interact with the same data, here operations are required to follow the consistency policy declared on data. Although ensuring consistency, this approach is not desired in many scenarios. For instance, the balance of a given bank account may be accessed by operations running at different consistency levels: *withdrawals* require coordination, but *deposits* do not. Furthermore, it is the developer that has to decide which is the most suitable consistency level that should be assigned for each region.

Another possible solution is Consistency Rationing [17]. It is a transaction paradigm that adapts the consistency guarantees at runtime. By requiring these consistency guarantees to be defined on data, rather than on transactions, it handles data according to its importance. This approach divides (rations) data into three consistency categories: the A category that provides serialisability and thus incurs high cost per transaction; the C category that provides session consistency (each client observes its own writes as long as their session lasts but they may not witness the updates made by another client), exhibiting low costs but where inconsistencies may arise; and, besides these two consistency guarantees, it also allows some data to have adaptable consistency in the B category, enabling it to switch between the semantics granted by both of the previous categories (depending on the specified policy).

These adaptive policies differ on the way they decide when to switch from one consistency semantic to the other. There are several strategies for adapting the consistency levels, such as:

Table 2.1: Overview of the existing solutions (regarding consistency semantics)

Solution	Input	Placement
Explicit Consistency [3]	invariants and post-conditions	service interface
QUELEA [36]	visibility and ordering properties	contracts
SIEVE [24]	invariants and merge semantics	database schema
DCCT [41]	consistency levels	data regions
Consistency Rationing [17]	data categories and policies	database schema
Our approach	*simple keywords*	*data declaration*

Table 2.2: Overview of the existing solutions (regarding integrity invariants)

Solution	Invariant specification	Invariant preservation
Explicit Consistency [3]	yes	yes
QUELEA [36]	no	-
SIEVE [24]	yes	no
DCCT [41]	no	-
Consistency Rationing [17]	no	-
Our approach	*lower/upper bounds*	[2]

- For any data item we have: the general policy that by monitoring the access frequency to data items, is able to calculate the probability of conflicting accesses, switching to serialisability if this probability is high enough; and the time policy that switches between consistency levels based on time (i.e., it runs under a certain consistency level until a given point in time and then switches);

- For numeric values we have: the fixed threshold policy that switches from one consistency semantic to the other according to the value of the data item (this static threshold might be difficult to specify); the demarcation policy that takes into account relative values regarding a global threshold; and the dynamic policy that analyses the update frequency of data items as well as their current values.

This solution seeks to reduce the overall costs by switching to cheaper consistency semantics whenever it is possible. Nevertheless, it introduces additional complexity into the development process since the programmer has to classify data into three consistency categories and specify the corresponding policies and all the integrity constraints (by annotating the database schema).

2.3.3 Summary

In general, the problem with control-centric strategies is that they impose on the application programmer the non-trivial burden of understanding the semantics of each operation in order to decide which properties should be assigned to each one of them. On both solutions presented in 2.3.1, reasoning is decentralized and a missing or a wrongfully written postcondition renders the specification incorrect.

All three data-centric models described in 2.3.2 propose an approach for specifying consistency properties based on the observation that correctness criteria and invariants are a property of data, not operations. As mapping high-level application consistency requirements to a combination of low-level consistency settings on operations can be quite challenging, these systems are simpler to describe and to reason about. However, as shown in Table 2.1 and Table 2.2, there are some aspects which our solution attempts to improve the existing data-centric approaches.

2.4 Identifying Operation Commutativity

Two operations commute if executing them serially despite their ordering against a given system state results in the same final state of that same system. In short, to find out if two operations commute it is necessary to check whether those two operations have the same side effects regardless of their ordering. Depending on the context, obtaining information concerning operation commutativity can be conveniently used for concurrency control or to find opportunities for parallelisation. Supposing that two operations do not commute and are simultaneously issued by different clients, depending on which operation is executed first, one of the clients may observe unexpected results. The design of the system may mitigate this outcome in different ways, but nothing can be done until the scenarios in which it arises have been identified.

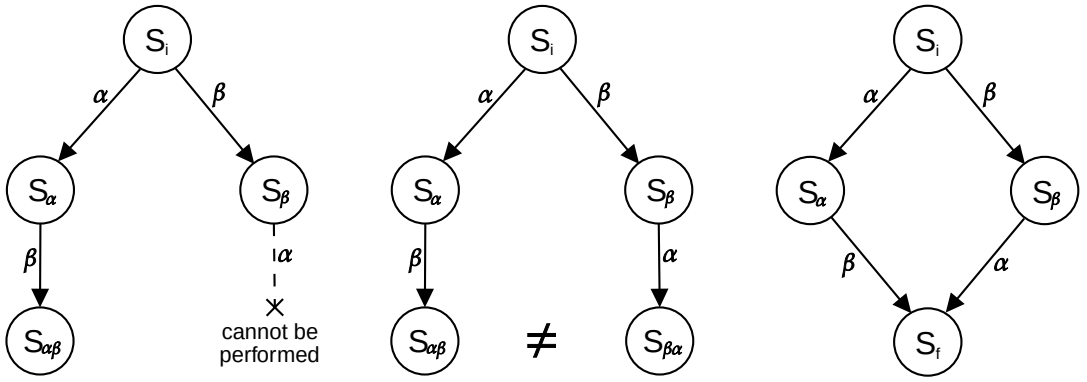
A recent work detailed in [15] describes some relevant concepts that are closely related to what our work helps to achieve: to be able to infer whether an operation can be performed locally (without global synchronisation between replicas) and whether two operations commute or not. They end up formalising this with the following concepts:

- Assuming an initial state θ where all system invariants are satisfied, a method call is *permissible* if its execution results in a post-state θ' where all system invariants are preserved. A method call is *locally permissible* if it is *permissible* at the replica in which it was requested. However, when a *locally permissible* method call is broadcasted to other replicas it is not necessarily *permissible* when it arrives.

- Two method calls *S-commute* (state-commute) if they generate the same final state when both are executed against a given initial state regardless of the order in which they are performed. Otherwise, they *S-conflict* (state-conflict) and synchronisation is needed in order to execute them in the same order across all replicas.

Yet, it is essential to understand how one can actually infer if two operations may conflict with each other (i.e., if they do not commute). There are several solutions regarding this topic, namely [2, 11, 30], however, given that we are working in the context of

[2]Since this is a more runtime-oriented subject, this topic will not be addressed in this thesis. However, this work provides tools that allow this goal to be reached.



(a) Neither of the properties is satisfied.

(b) α and β are diamond-connected but not diamond-equivalent.

(c) α and β are diamond-equivalent.

Figure 2.4: Commutativity properties (adapted from [10])

primitive values, we can follow a similar approach to the one used in [10] and use a SAT solver[3] to check operation pairwise commutativity.

As stated in [10], operation commutativity consists of both of the following properties that are depicted in Figure 2.4:

- *Diamond connectivity* - Two operations α and β are *diamond-connected* if both sequences between them ($\alpha\beta$ and $\beta\alpha$) can be performed from the same initial state. If two *diamond-connected* operations (Figures 2.4b and 2.4c) are concurrently issued, then the execution of one of them cannot make the system evolve into a state where the second operation cannot be carried out. On the other hand, if two operations fail diamond connectivity (Figure 2.4a), that is, at least one sequence is not possible from the initial state, and are simultaneously issued, it can result in unexpected errors due to the imposed order over those operations.

- *Diamond equivalence* - Two operations α and β are diamond-equivalent if they are *diamond-connected* and the final state that results from executing both sequences ($\alpha\beta$ and $\beta\alpha$) are equivalent (where intermediate states may differ). Therefore, if two *diamond-equivalent* operations (Figure 2.4c) are issued at the same time by different clients, it does not matter which is executed first. If two operations fail diamond equivalence (Figures 2.4a and 2.4b) and are issued concurrently, then the final state of the system after executing them is unpredictable.

All previous concepts and properties acted as a guideline for our commutativity analysis which will be described later in this document.

[3] A SAT (boolean satisfiability problem) solver seeks to assign a value to all variables of a given formula in order to conclude if it is satisfiable, that is, if it can be evaluated to true.

Chapter 3

Proposed Solution

In this chapter we describe the data-centric model that acted as the baseline for our work regarding the commutativity analysis between operations and we define the corresponding adjustments that were introduced to it to better fit our purpose (so that the programmer is able to specify the desired behaviour of a given replicated system using a simple input). Thus, the required modifications that the developer needs to add to the original source code in order to avoid concurrency errors (wrong states) are made clear. We also provide a brief overview of our solution as well as some considerations concerning the usefulness of the information generated by our compile-time analysis by showing how a replicated system can benefit from it.

3.1 The RC^3 Model

The RC^3 (Resource-Centred Concurrency Control) model is presented as an extension to programming languages, adding to the host language a set of type modifiers applicable to any variable declaration (class fields, local variables and parameters) or to the return type of methods. These modifiers allow the programmer to indicate which data items (the resources) contain values that are susceptible to concurrent manipulation and thus require consistency guarantees. In the context of this work, Java is the host language, whereby the type modifiers are implemented as annotations (mechanism for adding metadata information to source code).

In [28], the RC^3 model for concurrency control in shared memory environments was presented. To this end, the `@Atomic` keyword was defined in order to allow the programmer to identify which variables (named *atomic variables*) require the aforementioned consistency guarantees (acting as an automatic lock generation mechanism). In the RC^3 model, the unit of work is the method: all values accessed via an *atomic variable* in the

Listing 3.1: Bank account (atomic version)

```
1  class Account {
2
3    private float balance;
4
5    public Account(){
6      this.balance = 0;
7    }
8
9    void deposit(float amount) {
10     this.balance += amount;
11   }
12
13   void withdraw(float amount) throws OverdraftException {
14     if(this.balance - amount >= 0)
15         this.balance -= amount;
16     else throw new OverdraftException();
17   }
18
19   void accrueInterest(float interest) {
20     deposit(this.balance * interest);
21   }
22
23   void transfer(@Atomic Account to, float amount) throws OverdraftException {
24     withdraw(amount);
25     to.deposit(amount);
26   }
27
28   float getBalance() {
29     return this.balance;
30   }
31
32 }
```

body of a method share a consistency relation. Consequently, the model semantics ensure that all write accesses to this set of values are seen as an atomic operation for the rest of the system. This same model guarantees safety properties (such as strong atomicity, absence of data races and serialisability) and liveness properties (such as progress). These properties are secured at different stages of the compilation process, whereas the final generated code comprises a set of operations over locks that ensure that concurrent method executions will generate the same result as their serial execution. By recognising which data items require consistency guarantees (through the `@Atomic` keyword), RC^3 associates one lock to each atomic resource and imposes the lock acquisition to be performed before the first access to its corresponding resource in a method execution. Naturally, it also guarantees that these acquisitions never drive the system to deadlocked states.

Given the code presented in Listing 3.1, there is no guarantee that the account itself over which operations are applied (the *this* parameter) is atomic. However, if the entity that manages these same accounts (the bank) has each account identifier mapped into its corresponding atomic account, whenever an operation is requested it only has to obtain the corresponding atomic bank account(s) it requires and carry it out. This way, a *transfer*

Listing 3.2: Bank account (replicated version)

```
1  class Account {
2
3    @Replicated @Range(from = 0) private float balance;
4
5    public Account(){
6      this.balance = 0;
7    }
8
9    void deposit(@Range(from = 0) float amount) {
10     this.balance += amount;
11   }
12
13   void withdraw(@Range(from = 0) float amount) throws OverdraftException {
14     if(this.balance - amount >= 0)
15         this.balance -= amount;
16     else throw new OverdraftException();
17   }
18
19   void accrueInterest(@Range(from=0) float interest) {
20     deposit(this.balance * interest);
21   }
22
23   void transfer(Account to, @Range(from=0) float amount)
24       throws OverdraftException {
25     withdraw(amount);
26     to.deposit(amount);
27   }
28
29   float getBalance() {
30     return this.balance;
31   }
32
33 }
```

operation, for instance, might be carried out considering both accounts (*this* and *to*) as atomic accounts. Consequently, the model semantics ensure that *transfer* operations that operate upon the same accounts are performed in mutual exclusion in order to guarantee the consistency of their corresponding balances.

The RC^3 model also generates an intermediate code that comprises all the variants each method can possibly have (those that the programmer did not clearly define). For example, if the programmer did not state that the parameter of a given method is atomic, its argument can be either atomic or not. Hence, the model injects both method variants into the original code, i.e., multiple versions of each operation will be added to the source code in order to reflect all possible scenarios. Returning to the *transfer* example, the programmer should clearly define that the account to which a certain amount will be added (the *to* parameter) must be seen as an atomic resource. Otherwise, it can be either atomic or not, which could lead the system into incorrect states since the intermediate code generated by the RC^3 model for this concrete example would cover both scenarios.

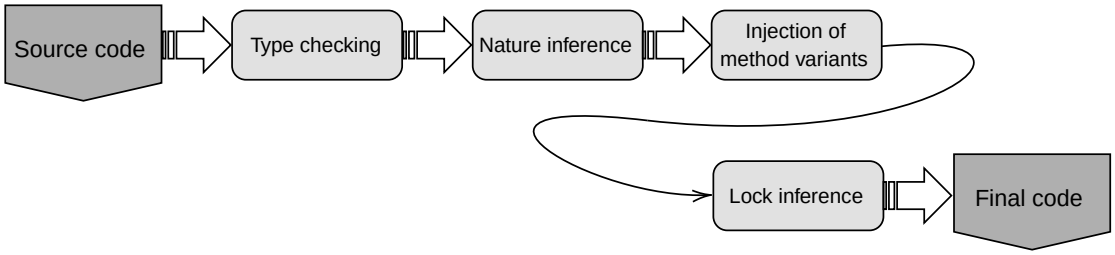


Figure 3.1: RC^3's pipeline of compilation stages

3.2 Addressing Replicated Systems

To address distributed memory environments (replicated systems), we added the `@Replicated` keyword, which allows the developer to express which parts of an application state are replicated by (potentially) multiple computers. We also added a new annotation - `@Range` - to define both lower and upper bounds for primitive data types to handle possible system invariants. Listing 3.2 illustrates how this model can be applied to implement a bank account whose balance is replicated and only defined in $\mathbb{N}_0$. Thus, if the banking service has multiple servers that replicate accounts in several parts of the globe, it should be possible to perform some operations concurrently on different replicas, such as *deposits* into the same account or *withdrawals* from different accounts, but it should not be possible to make two concurrent *withdrawals* from the same account. Our goal is to derive these properties statically, at compile time, from the simple keywords specified at data declaration.

Using a data-centric approach allows to centralise developer reasoning and automated analysis over data declaration. In the context of this thesis, our analysis is focused on the pairwise commutativity of methods from a given application (the unit of work of the RC^3 model). In other contexts, other static analyses may aim to guarantee different properties such as being impossible to change an atomic or replicated value without resorting to the compiler. A type system, whose definition is beyond the scope of this thesis, ensures that the annotations have impact over the variable types - the *t* type is different from the `@Atomic` *t* type and both are different from the `@Replicated` *t* type.

3.3 Solution Overview

The solution presented below fits into a pipeline of compilation stages, where its preceding phases already ensure that the code is correctly typed and that all inference to replicated variables has already been performed (see Figures 3.1 and 3.2).

Our solution, in turn, can be split into two main stages. Initially, it is necessary to understand how each method influences the state of a given replicated variable. Thus, it is necessary to analyse each method individually and extract the changes to the state of those same replicated variables that it causes. This way, all operations that manipulate

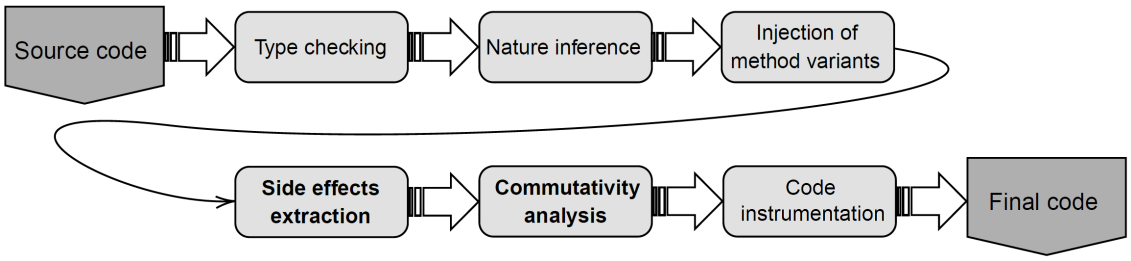


Figure 3.2: Where our solution fits into the pipeline of compilation stages

replicated variables are analysed in order to extract their corresponding side effects. More concisely, at the end of this step, all the state modifications of the replicated variables that each method may provoke are collected. However, for our purpose of commutativity analysis, only the writing effects are relevant since topics such as inconsistencies that may arise from read operations are beyond the scope of this work and are already resolved by the RC^3 version in the shared memory context that ensures mutual exclusion when applying these operations in a single replica.

Secondly, the operation pairwise commutativity check is performed based on their corresponding side effects that were extracted in the previous phase. As stated before, in order to proceed with the commutativity analysis of these side effects, we based our approach on the technique described in [10], where the problem of checking commutativity of two operations is reduced to a constraint solving problem. Thus, for each scenario, it is mandatory to build a constraint system that not only takes into a account the state changes each operation generates as well as the domain of the replicated variables they manipulate. After all possible scenarios have been evaluated it is then possible to identify the set of operations that conflict with each other (i.e., do not commute) that must be payed particular attention at runtime.

The last compilation stage of the pipeline (code instrumentation) was not addressed in this work. Nevertheless, it will be mandatory to integrate the results our compile-time analysis generates into the original code by inserting calls to a runtime responsible for ensuring exclusivity when it is required.

Our main focus is to generate, at compile time, all the necessary information regarding operation commutativity based solely on replicated data declaration (as depicted in Figure 3.3). The result of our work may, in turn, be used at runtime avoiding unnecessary computations seeking to offer the best possible performance.

3.4 Relevant Considerations

Firstly, this work is intended to identify operations that cannot be performed locally on a given replica without synchronising with the remaining ones. Therefore, recalling our running example of the bank account whose balance is replicated, a *withdrawal* operation cannot be performed locally as it may threaten the non-negativity invariant of the

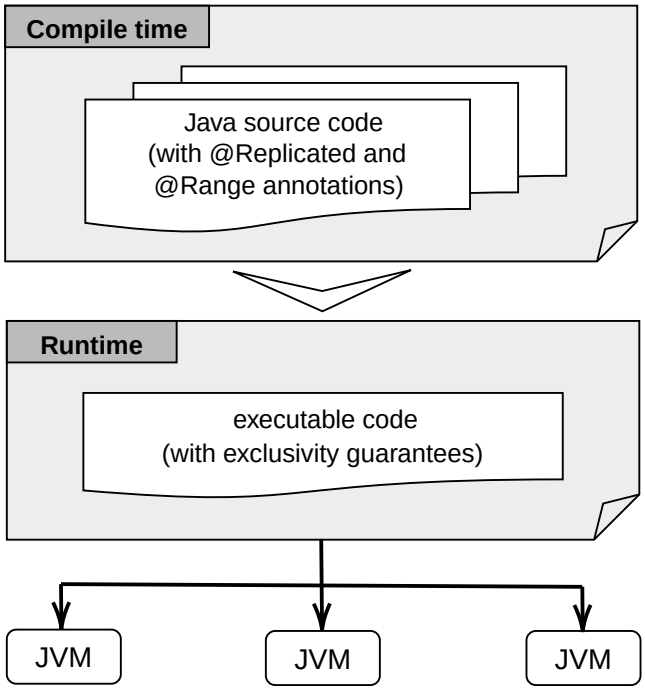


Figure 3.3: General concept of the proposed solution

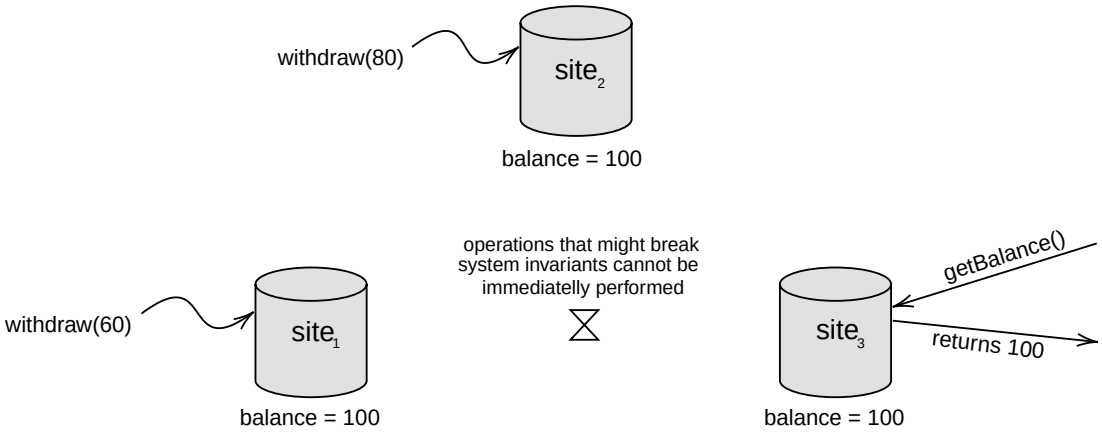


Figure 3.4: Operations that can and cannot be performed locally in a replicated system

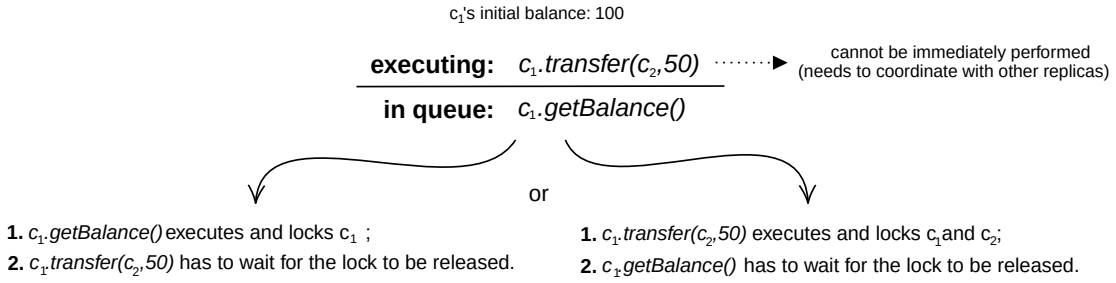


Figure 3.5: Example of how a given replica deals with concurrently issued operations

balance (nothing prevents another replica from executing another *withdrawal* simultaneously). However, as shown in Figure 3.4, any *balance inquiry* may very well execute locally since it has no side effects regarding the replicated state. Obtaining this information is straightforward from evaluating the side effects of each method that are extracted at the first stage of our solution.

Furthermore, it is necessary to understand how the different replicas of a given replicated system should behave individually when required to perform several operations simultaneously taking into account the information generated by our analysis. Considering the example depicted in Figure 3.5 where we just look at a single replica of a given system, when an operation is requested from a particular replica, it is necessary to understand whether that same operation can perform without having to carry out a system coordination phase or it conflicts with any operation that is performing at that time (runtime must ensure an efficient way to achieve this coordination between replicas, which can be achieved through vector clocks, for example). To assess if a given operation can execute locally, i.e., it does not threaten any system invariant, the system must consider the information that was previously generated at compile time and check if that same operation does not conflict (commutes) with itself.

It should be noted that, for the model under consideration, the replicated variables declared with the `@Replicated` annotation are also *atomic variables*. The usefulness of this approach is that operations running concurrently on the same machine continue to suffer from this atomic semantics including those that were remotely requested and performed locally. In this particular case, if by the time the *getBalance* operation arrives the *transfer* operation is on hold (awaiting for coordination to take place), the former takes place and locks c_1. Even if the *transfer* operation is ready to execute while the *getBalance* operation is being executed, it cannot execute because one of the objects it depends on is locked. If by the time the *getBalance* operation arrives the coordination phase is already over and the *transfer* operation is being performed, even if it does not endanger system invariants, it cannot execute because c_1 is locked. If the balance inquiry operation was performed on an account other than c_1 or c_2, it could be executed even if the *transfer* operation was in progress.

Although the final compilation stage in Figure 3.2 (code instrumentation) was not ad-

Listing 3.3: Code instrumentation for our running example

```
class Account {

  ...

  void transfer (Account to, float amount) throws OverdraftException {
    Runtime.run_global(transfer,this,to);
    withdraw_internal(amount);
    to.deposit_internal(amount);
  }

  void withdraw(@Range(from = 0) float amount) throws OverdraftException {
    Runtime.run_global(withdraw,this);
    ...
  }

  void withdraw_internal(@Range(from = 0) float amount)
    throws OverdraftException {
    ...
  }

  void deposit(@Range(from = 0) float amount) {
    Runtime.run_local(deposit,this);
    ...
  }

  void deposit_internal(@Range(from = 0) float amount) {
    ...
  }

  ...

}
```

dressed in this work, the original source code must be modified by injecting a runtime call at the beginning of each method differentiating *locally permissible* operations from those that require coordination. When instrumenting the code it is necessary to distinguish between top-level operations from those that are called within other methods. With this in mind, each method needs to have two distinct versions (as depicted in Listing 3.3 for the *transfer* operation). Thus, when a *transfer* operation is issued, the runtime is only called once and the information generated previously at compile time is only evaluated once. To handle transactions and using the *transfer* operation that delegates its functions to other methods as an example, we only check the top-level operation (*transfer*). Otherwise, if we examined its elementary operations, we would conclude that the *withdraw* operation would need strong consistency to execute while the *deposit* operation would not. In practice, when a *transfer* operation is requested, the runtime is only called once and the information produced by the previous commutativity analysis is only queried for that same method. This information already covers the operations that are called by this method so it is guaranteed that the strongest consistency level that is required is applied. If we do not differentiate methods into two versions, we would no longer have transactions and

would only have elementary operations and would have three unnecessary runtime calls. Given that our goal is to decrease the latency of replicated systems, if the decision has already been made for the *transfer* operation at the first runtime call, there is no need to recheck that same information for its elementary operations (*deposit* and *withdraw*).

Chapter 4

Implementation

This chapter describes the implementation of both stages of our solution, explains all of its steps and also discusses some relevant details. We start by showing the way we identify how each operation of a given service affects its replicated state. Subsequently, we not only clarify our approach to infer the operation pairwise commutativity properties as we also point out some design considerations that allowed us to broaden the space of non-conflicting operations.

The solution described in the following pages was developed in the Java language and in the context of the Eclipse JDT Core component that provides an API for navigating the Java element tree, generating an intermediate representation of it.

4.1 1st Stage - Side Effects Extraction

Using the JDT Core plugin that includes a Java model that provides an API for navigating the Java AST[1], it is possible to manage all elements of a given Java code. Taking advantage of the visitor design pattern, we developed a visitor to go through all the method variants present in the intermediate code generated by the RC^3 model. This method visitor supports some types of nodes from literals, variables and field accesses to unary operations (prefix and postfix expressions) and binary operations (infix expressions). Some more complex nodes, such as loops, were not addressed in this work.

4.1.1 Converting JDT Core Expressions

As previously stated, JDT Core expressions are an intermediate representation of the corresponding nodes that the Java element tree comprises. Therefore, we need to convert

[1] An Abstract Syntax Tree is an internal data structure that represents the structure of a source code and where each node denotes a construct occurring in it.

Operation	Effects List
deposit	[=(this.balance, +(this.balance, amount))]

Table 4.1: Side effects of the *deposit* operation

JDT Core expressions that our visitor traverses into our own notation. For each visited node, we generate a new representation of the corresponding language constructor that appears in the AST and add all the information we consider necessary. For instance, if a method manipulates a given variable, it is required to understand if it was declared as replicated or not (according to our model). To that end, at each step (visited method) we must have access to both current class and current method in order to evaluate its scope[2]. Thus, we are able to generate our own notation which is independent from the one Eclipse offers, adding all the information we need to proceed to the next step since the syntax of these side effects must be sufficiently rich to proceed with the commutativity analysis.

4.1.2 Generators and Effectors

Considering a method as a set of expressions (its operations), we may discard those that do not manipulate the replicated state (non-replicated expressions) since they do not add nothing new to our purpose. As previously stated, we are only interested in write operations, that is, operations that change the replicated state. As we are working over primitive data types, an assignment to a local variable (named *assignment*) will never change the replicated state of a given service. In this sense, the only expressions that need to be analysed are assignments to class fields, named *updates*.

So, for each individual *update*, we have to figure out if it covers some replicated node (this is also applicable to unary expressions like x++ (where x is replicated) since it converted to an *update* node after it is traversed for the first time). More specifically, whether its left-hand side is a replicated field (declared with the `@Replicated` keyword) or the expression on its right-hand side handles some replicated node. For example, a binary operation handles replicated nodes if one of its operands manipulates replicated nodes. If the expression being analysed is significant to our analysis, we need to attach it to the current method information. To that end, we created the notion of *method complement* where we can store extra information about its corresponding method. Each method may have attached to itself several extras (identified by a certain keyword regarding its purpose), but in this case, it is enough to store a single list of replicated expressions - an *effects list* - as well as the set of replicated variables that the method manipulates. An example of the *effects list* of the *deposit* operation is shown in Table 4.1.

Generally speaking, operations can be split into: *generators*, that are side effect free and can be applied on any replica; and *effectors*, that contrarily to the previous ones, have

[2]However, given that we are in the context of primitive data types, any local variable declared as replicated is irrelevant for our analysis.

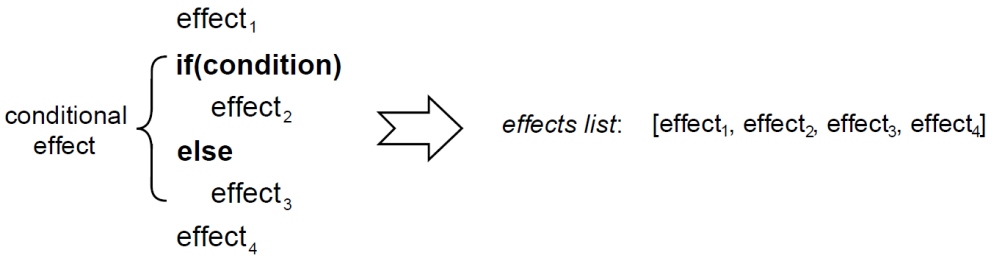


Figure 4.1: Generic example of a conditional effect

Operation	Effects List
deposit	[=(this.balance, +(this.balance, amount))]
withdraw	[=(this.balance, -(this.balance, amount))]

Table 4.2: Side effects of *deposit* and *withdraw* operations

impact over the replicated state and, for that reason, may need synchronisation. Evidently, only *effectors* need to be considered in our commutativity analysis. Recalling our running example, while *deposits*, *withdrawals*, *transfers* and *interest accruals* have impact over the replicated state, a *balance check* does not (it is just a read operation).

4.1.3 Conditional Effects

As depicted in Figure 4.1, a conditional effect comprises, as the name implies, all effects that are within the same conditional block. Thus, the *effects list* of the corresponding method depends on the conditional statement of that same block.

We took a conservative approach and, in these cases, we consider all side effects regardless of their branch, that is, we concatenate all effects within a conditional block. Therefore, the side effects of the *withdraw* operation, following our approach, are the ones shown in Table 4.2.

4.1.4 Decomposing Operations

Before adding any replicated expression to the aforementioned *effects list*, each one of them goes through a decomposition process. In order to broaden the commutativity space of non-conflicting methods, we catch all the compound binary operations or, in other words, we capture by recursion all the binary operations that act as operands of another binary operation.

Recalling the toy banking application, an *interest accrual* performs a multiplication and an addition, and thus this operation does not commute with, for example, a *deposit* operation. However, if we assume that the multiplication operation is computed locally at the replica where the operation was issued (considering it as a mere constant), it can

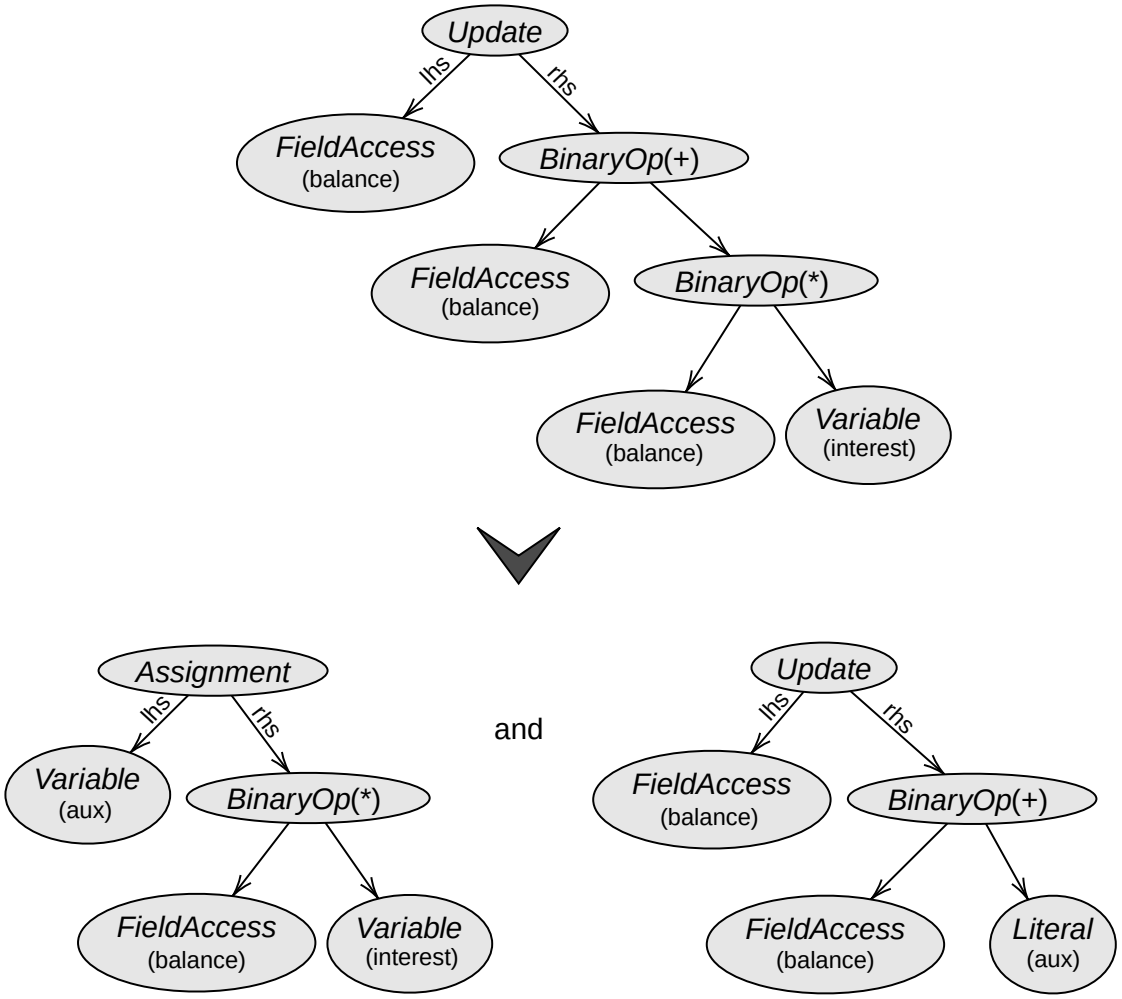


Figure 4.2: Decomposition process for the *interest accrual* operation

Operation	**Effects List**
deposit	[=(this.balance, +(this.balance, amount))]
withdraw	[=(this.balance, -(this.balance, amount))]
accrueInterest	[=(aux_1, *(this.balance, interest)), =(this.balance, +(this.balance, aux_1))]

Table 4.3: Side effects of *deposit*, *withdraw* and *accrueInterest* operations

be classified as a *generator operation* and the addition operation is the one that is disseminated to the other replicas (*effector operation*), being equivalent to a *deposit* operation and reducing the number of methods that conflict with each other. Thus, *interest accrual* commutes with *deposit* (two addition operations). For each local operation, we create an auxiliary (non-replicated) variable with a unique identifier and build an *assignment* node

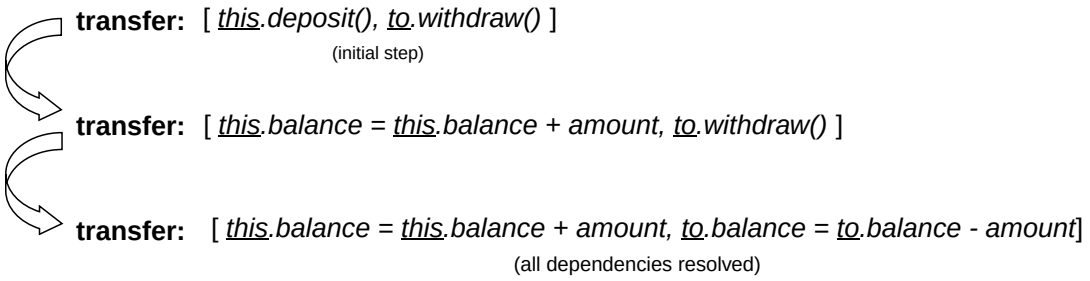


Figure 4.3: Handling method dependencies of the *transfer* operation

Operation	Effects List
deposit	[=(this.balance, +(this.balance, amount))]
withdraw	[=(this.balance, -(this.balance, amount))]
accrueInterest	[=(aux_1, *(this.balance, interest)), =(this.balance, +(this.balance, aux_1))]
transfer	[=(this.balance, -(this.balance, amount)), =(to.balance, +(to.balance, amount))]

Table 4.4: Side effects of *deposit*, *withdraw*, *accrueInterest* and *transfer* operations

over that same variable with the corresponding right-hand side expression (as can be seen in Figure 4.2). The side effects of an *interest accrual* operation that result from following the previous approach can be found in Table 4.3.

4.1.5 Method Dependencies

During this process of collecting side effects, when a method calls another method that handles replicated variables, it is necessary to insert the effects of the latter into the *effects list* of the former (e.g. *transfer* depends on *withdraw* and *deposit* operations).

To this end, and since we cannot handle method dependencies as we are collecting effects because there is no guarantee that a particular visited method does not depend on others that have not yet been visited, that is, their effects have not yet been extracted, a new visitor was developed to traverse the dependencies of method calls. For the method being visited, its dependencies are traversed and the necessary procedures are performed so that its corresponding effects are collected. After all its dependencies are resolved, a later visited method that depends on the previously mentioned one will not need to resolve its dependencies since they are already resolved (avoiding unnecessary steps).

For instance, when the *transfer* method is being visited (as shown in Figure 4.3), we visit its dependencies (namely *withdraw* and *deposit* operations). If any of these operations do not have their dependencies resolved, they are settled in this step (in this particular

Operation	Parameters	Effects List
deposit	(this)	[=(this.balance, +(this.balance, amount))]
withdraw	(this)	[=(this.balance, -(this.balance, amount))]
accrueInterest	(this)	[=(aux_1, *(this.balance, interest)), =(this.balance, +(this.balance, aux_1))]
transfer	(this, to)	[=(this.balance, -(this.balance, amount)), =(to.balance, +(to.balance, amount))]
getBalance	(this)	[ ]

Table 4.5: Side effects of each individual operation of our running example

case, since none of the operations have dependent calls, it is sufficient to use their initial effects directly). If another method relies on the *transfer* method, it could already use its corresponding effects because all the dependencies of this method would have already been resolved and its *effects list* would already reflect those same dependencies.

To reflect dependencies on the side effects of a given operation, we do not copy the effects of its dependencies directly. Instead, we clone the *effects list* and change the instance that invokes it if it is required. For example, when considering the *effects list* of the *withdraw* operation when resolving the dependencies of the *transfer* operation, its target would have to be changed from *this* to *to* and those changes must be reflected on the *effects list* of the latter (see Table 4.4).

4.1.6 Output

Even though we are working over primitive types, a relevant point to keep in mind is the following: at this stage, it is necessary to distinguish between operations that are performed over the same object from others. For example, two simultaneous *withdrawal* operations on the same account imply strong consistency. However, if executed on different accounts, applying weak consistency will suffice. In order to distinguish between these two cases, the *effects list* of a method *m* has to be parameterised, i.e., it has to consider *m* parameters as parameters that have a replicated part of their state and that same part is accessed in the *m* body. This *effects list* can be seen as a function whose parameters are the replicated parameters accessed inside the method body plus the *this* parameter.

That being said, by completing this first stage for our running example, its final output is something similar to what is shown in Table 4.5 where, for each method, the list of effects over replicated variables is extracted, as well as their respective parameters. One question that may arise is how to deal with the constructor method. In this particular example, it initialises a non-static private field (*balance*) and so, a commutativity analysis on it would not make any sense since only after it executes the field becomes available, that is why it was discarded.

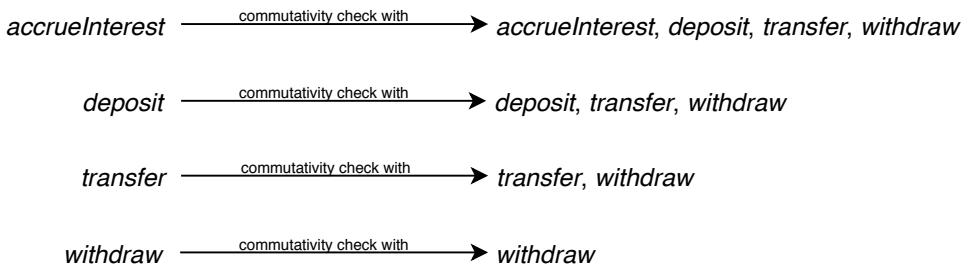


Figure 4.4: Methods with which each operation checks its commutativity

The *getBalance* operation has no side effects since it is just a read operation and does not apply any changes over the replicated state. In this particular example, all operations handle the same replicated variable (*balance*), however, in an example other than this, not all operations would have to handle the same replicated variable.

Throughout this phase, in addition to storing information about the replicated variables that each method manipulates, information on the methods that manipulate each of the replicated variables is also collected, thus easing the commutativity analysis itself that will only be elaborated between pairs of methods that operate over common replicated variables (otherwise, they commute), where for each pair, all possible parameter combinations have to be considered. Thus, even if at compile time we do not know the instance in question, we can infer commutativity for all possible parameterisations.

4.2 2nd Stage - Commutativity Analysis

As stated earlier, we reduced the problem of inferring operation pairwise commutativity to a constraint solving problem. In this sense, we took advantage of the JaCoP (Java Constraint Programming) solver, a Java language library, to detect which operations conflict with each other by formulating, for each pair of methods that manipulate the same replicated state, a constraint system that reflects their side effects as well as the domains of the variables over which they operate.

4.2.1 Constraint System

Given the fact that we are facing a context of primitive variables, the constraint system that determines if two methods commute is naturally built by mapping each side effect of both methods into a constraint (considering each method assignment as an equality).

As mentioned before, our commutativity analysis only covers pairs of methods that change the state of the same replicated variables (those that may interfere with each other) and these methods are analysed according to their lexicographic order to avoid repeated and unnecessary analysis. A method may conflict with itself by threatening system invariants when concurrently executed (for example, a subtraction operation in $\mathbb{N}_0$ does not commute with itself), that is why each method of a given application has to check if

it commutes with itself (as shown in Figure 4.4). Following this approach, each operation only checks its commutativity with methods whose signature does not lexicographically precede its own and, therefore, the same pair of operations is not evaluated twice.

The operation pairwise commutativity is then verified by generating two equation systems (one for each sequence). In order to establish the right constraints regarding the two possible sequences between two operations, it is necessary to generate some solver variables that recreate the behaviour of the variables declared in the original code, taking into account the lower and/or upper bounds that were defined by the programmer with the help of the `@Range` keyword.

In our solution, we handle the generated solver variables in three distinct ways taking into account the kind of variables they recreate:

- *Left-hand side variables* - for each assignment in the body of a method, it is necessary to create a new solver variable concerning the variable to which a given value will be assigned. These variables must be unbounded, otherwise, if they had the corresponding domain of the variable they recreate, they would influence the final result (the values assigned to the right-hand side variables) and it would not be possible to infer whether a given operation could endanger the system invariants.
- *Method parameters* - the generated solver variables for method parameters must be stored in a way that an operation uses the same value as a parameter in the two possible sequences. Otherwise, they could take different values in both cases, leading to an incorrect result.
- *Pre-state* - initially, it is also necessary to create solver variables that reflect all the possible initial values of a given replicated variable that the methods in question manipulate. Naturally, as new assignments to the same variable emerge in a given sequence of operations in a method body, the values to be considered will not be the ones that these variables refer to but rather those of the left-hand side variables that were most recently used.

Not to be so strict in analysing operation commutativity, it is of the utmost importance to address all the possible instance combinations upon each operation is performed. More specifically, we need to be able to distinguish operations that are executed over the same object from the ones that do not. To that end, each case has to be identified and individually analysed. For instance, two simultaneous *withdrawals* from the same account do not commute. Contrarily, two concurrently executed *withdrawals* over different accounts do commute (in these cases where the two operations do not change the state of the same account we do not proceed with our analysis since these methods commute). Summarily, our commutativity analysis only considers pairs of methods that manipulate replicated variables in common but, for each of these pairs, it also considers all the possible configurations for their corresponding parameters (as discussed in the previous stage - Table 4.5).

Evidently, this instance checking can only be performed at runtime. However, a constraint system can be built and solved for all possible cases at compile time, making the runtime task more efficient where after this information concerning conflicting methods is generated, the result can be easily read as it has already been computed. To this end, as we are analysing each pair of methods, we clone their *effects list* and parameterise it with the corresponding configuration we are evaluating.

$$\begin{cases} lhs_1 = this.balance + Account.deposit.amount_1 \\ lhs_2 = lhs_1 + Account.deposit.amount_2 \end{cases} \tag{4.1}$$

$$\begin{cases} lhs_3 = this.balance + Account.deposit.amount_2 \\ lhs_4 = lhs_3 + Account.deposit.amount_1 \end{cases} \tag{4.2}$$

To start off with a really simple example, we will consider the pair of operations (*deposit*,*deposit*). Having Table 4.5 as a reference, we can conclude that this operation commutes with itself even if the account is the same since an addition will never threaten the non-negativity of its balance. Both sequences presented in (4.1) and (4.2) refer to the case where both *effects list* are similarly parameterised (that is, the accounts coincide) and they perfectly demonstrate how the generated solver variables are handled: all solver variables that recreate left-hand side variables are distinct; as both method parameters may differ (two *deposits* with different amounts), two different solver variables with a lower bound of zero (range invariant) are generated and are used by the same operation in both sequences; and, as the first operation of each sequence accesses the current state of the *balance* field, a new solver variable is generated to recreate that same state (non-negative variable, assuming that the previous *balance* state satisfies its invariant), however, the second operation in both sequences no longer uses this variable but the most recent left-hand side variable whose value corresponds to the state after executing the first operation (*deposit*).

$$\begin{cases} lhs_1 = this.balance - Account.transfer.amount_1 \\ lhs_2 = to.balance + Account.transfer.amount_1 \\ lhs_3 = lhs_1 + Account.deposit.amount_1 \end{cases} \tag{4.3}$$

$$\begin{cases} lhs_4 = this.balance + Account.deposit.amount_1 \\ lhs_5 = lhs_4 - Account.transfer.amount_1 \\ lhs_6 = to.balance + Account.transfer.amount_1 \end{cases} \tag{4.4}$$

Considering the pair of operations (*deposit*,*transfer*), they only conflict if the parameter *this* from which the effects of the *deposit* method depend on is equivalent to the *this* parameter of the *transfer* method - the constraint system for this particular case is shown in (4.3) and (4.4). In this case, the equation system presented in (4.3) refers to the case

where the operation *transfer* occurs first and the equation system in (4.4) refers to the case when it is processed second. If the *transfer* operation is executed first, the *balance* might not be high enough to proceed with the operation. However, if *deposit* is performed first, the previous *transfer* operation might be completed. For instance, considering that our current balance is 100 €, and we want to add 50 € to it and transfer 150 € to another account. Our balance after these operations have been executed depends on the order by which they were performed. If the *transfer* operation precedes the *deposit* operation, we do not have enough balance and the *transfer* operation fails and the only operation that is performed is the *deposit* one (final balance is 150 €). Otherwise, if the *deposit* operation precedes the *transfer* operation, both succeed (final balance is 0 €). These operations are not *diamond-equivalent* and thus, do not commute.

The equality constraints imposed in each of the previous examples are not sufficient to infer the commutativity between their operations. Two operations only commute if the final state in both scenarios is equivalent. Recalling the previous pair of methods (*deposit*,*transfer*) example: for these operations to commute both lhs_3's and lhs_5's value have to be equal. In this sense, we need to add the $lhs_3 \neq lhs_5$ constraint to the previous constraint system to find out if there is a given solution where both final states do not match. This verification is made between lhs_3 and lhs_5 since these are the solver variables that refer to the final value of the common replicated variable in both sequences (lhs_6 refers to the *balance* of the *to* account that we considered different from the *this* account). If the solver finds any solution for this constraint system, this result tells us that the two operations do not commute because the result is not independent of the order of execution of the considered operations. Otherwise, they commute.

However, imposing the previous constraint is still not enough. In order to preserve the system invariants when executing both sequences, we cannot only cover the inequality between the final state of the common replicated variables. We must also check if both operations do not introduce an incorrect state in the system. We need to verify if all values that might be assigned to each left-hand side variable that refer to the same replicated variable each operation handles are in the range of those same replicated variables. If there is any solution where a left-hand side has a value that is out of bounds, both operations conflict with each other. That being said, in the previous (*deposit*,*transfer*) example, the $lhs_3 \neq lhs_5$ constraint must be replaced with the following constraint:

$$lhs_3 \neq lhs_5 \quad \vee \quad lhs_1 < 0 \quad \vee \quad lhs_3 < 0 \quad \vee \quad lhs_4 < 0 \quad \vee \quad lhs_5 < 0 \tag{4.5}$$

To sum up, if the solver finds a solution that satisfies the equality constraints imposed in (4.3) and (4.4) and at least one of the constraints imposed in (4.5), operations do not commute as they either do not reach the same final state or one of them leads the system to an inconsistent state (i.e., some invariant is violated). Obviously, and as previously discussed, the solver would find a solution for the constraint system that refers to the previous (*deposit*,*transfer*) example. However, there are some parameterisations of their

corresponding *effects list* that make them commute, namely when all its parameters are different.

4.2.2 Solver Primitives

The JaCoP library, as specified in its documentation [18], provides primitives to define finite domain variables (FDVs) as well as floating point variables (FPVs), constraints over these variables (such as equality, inequality and conditional constraints) and search methods.

Annex I shows how we can define all the variables we need. It also presents the set of constraint primitives provided by the JaCoP library that were considered when setting up the constraint systems required for the operation pairwise commutativity analysis (these primitives include basic operations and basic relations between variables). These constraints are imposed over a *store* that is the JaCoP's way of representing the constraint model for which a constraint solver will try to find solutions to, i.e., it will try to assign values to the variables defined in it so that all imposed constraints are satisfied.

For arithmetic constraints, the subtraction operation is not supported. However, it is easily defined using the addition operation where the domain of the variable to be subtracted must be set accordingly. This implies that to translate a *transfer* operation, for instance, into its respective equations we need to generate two solver variables to recreate the *amount* parameter (one for each operation - addition and subtraction). Therefore, it is also necessary to ensure that the absolute value of the variable recreating the *amount* to be withdrawn is identical to the value of the variable recreating the amount to be deposited.

In short, all we have to do is to distinguish between the different operators of the expressions being evaluated and impose the constraint to which the latter corresponds. For instance, recalling the bank account example and considering the step in which the commutativity analysis is carried out between two *deposit* operations over the same account: since both operations handle floating point variables and both perform an addition to the previous balance, we only need to take advantage of the *PplusQeqR* primitive JaCoP provides to recreate both possible sequences and *Or*, *PneqQ* and *PltC* primitives to impose the remaining constraints (as can be seen in (4.6), (4.7) and (4.8), assuming that initial state variables (*this.balance*), left-hand side variables (lhs_1, lhs_2, lhs_3, lhs_4) and those that recreate method parameters ($amount_1$, $amount_2$) have already been correctly created regarding their domain).

$$\begin{cases} PplusQeqR(this.balance, amount_1, lhs_1) \\ PplusQeqR(lhs_1, amount_2, lhs_2) \end{cases} \tag{4.6}$$

$$\begin{cases} PplusQeqR(this.balance, amount_2, lhs_3) \\ PplusQeqR(lhs_3, amount_1, lhs_4) \end{cases} \tag{4.7}$$

$$Or(PneqQ(lhs_2, lhs_4),\ PltC(lhs_1, 0),\ PltC(lhs_2, 0),\ PltC(lhs_3, 0),\ PltC(lhs_4, 0)) \tag{4.8}$$

deposit(this)	*[withdraw(this), transfer(this,_)]*
withdraw(this)	*[deposit(this), withdraw(this), accrueInterest(this), transfer(this,_), transfer(_,this)]*
accrueInterest(this)	*[withdraw(this), transfer(this,_)]*
transfer(this,to)	*[deposit(this), withdraw(this), withdraw(to), accrueInterest(this), transfer(this,_), transfer(_,this), transfer(to,_)]*
getBalance(this)	*[ ]*

Table 4.6: Conflicting operations for the bank account example

4.2.3 Search Step

After specifying the constraint model, that is, after formulating the system of equations that represents the scenario under analysis, it is necessary to find a solution that satisfies those same conditions. Although JaCoP library provides primitives that allow us to search for all existing solutions, in our particular case, it is sufficient to find a single solution to infer that two operations do not commute according to a given configuration of replicated parameters. If there is a case where either the replicated variables do not have the same value at the end of the two sequences of operations or the system invariants are violated during that same process, those operations can never commute.

As the ideal search method depends on the imposed constraints and the way it assigns values to the solver variables and the order in which these assignments are performed, to find a single solution we adopted the default depth-first search algorithm provided by JaCoP. This algorithm converts its search space into a search tree where each of its nodes represents the assignment of values to the variables defined in the constraint model (following nodes will adjust the domain of its variables according to this assignment). Naturally, once a given node breaks a given system invariant, the branches that will naturally continue to not satisfy all imposed conditions will not be analysed. For integer variables, the default option is to start assigning values to variables starting with the minimum values in their domain and where the variable with the smallest domain is selected first. For floating point variables, a domain split search (bisection) is used. In this case, the search method divides the domain of the variables in half and proceeds its search for solutions using the first half first.

Since the search steps differ from one type of variables to another (finite domain variables and floating point variables), if the application code to be analysed contains replicated variables that cover both cases, these must be resolved in separate steps.

4.2.4 Output

At the end of this stage, all the information regarding operation pairwise commutativity will have been generated. As can be seen from Table 4.6, for each individual operation of a given application code, we not only provide the list of the methods with which it conflicts but we also indicate the parameterisations that lead one operation to conflict with another. For example, while a *transfer* operation conflicts with a *deposit* operation when the bank account we intend to transfer from is the same as the one we desire to deposit to (a subtraction and an addition over the same replicated variable might break the balance non-negativity system invariant), a *transfer* operation does conflict with a *deposit* operation when the bank account we intend to transfer to is the same as the one we desire to deposit to (two additions over the same replicated *balance*) since the system invariant can never be violated by these operations.

To represent the previous information, we map each of the evaluated pairs of operations into a list containing all their conflicting parameterisations. For instance, the (*accrueInterest*,*transfer*) pair is mapped into *[([0],[0,0]),([0],[0,1])]* and (*deposit*,*deposit*) into an empty list. Thus, considering the information provided by our compile-time analysis we are able to distinguish between operations that can be performed under weak consistency and those that, inevitably, have to be performed under strong consistency semantics. At runtime it is possible to infer if a given method m can be executed locally by verifying if that same operation commutes with itself, that is, $(m,m) \rightarrow []$, or if the replica to which the operation was requested needs to start a synchronisation process to be able to perform it. With this information it is also possible to infer if a given operation might be executed while others are waiting for synchronisation to take place by checking if it commutes (i.e., does not conflict) with any operation executing at that moment and all operations in queue.

Chapter 5

Evaluation Methodology

In this chapter we present our experimental work and discuss its corresponding outcomes. Our goal is to show that our data-centric concurrency control solution is not only easy to apply because of the simplicity of the modifications to the original source code it requires, but that it also incurs in low compile time overheads to achieve the intended results. That being said, to evaluate our solution we considered two main metrics: the accuracy of the output that our analysis produces and the compile time overhead it incurs to achieve the expected results. By assessing these two metrics we intend to show the feasibility of adopting our solution when setting up a given replicated system.

All tests were conducted in a machine that had Java 8 installed and had the following specifications: Intel Core i7 2.4GHz dual-core processor with 8GB of memory.

5.1 Correctness

The primary focus concerning the correctness of the developed solution was to ensure that, for each operation of a given application code, all its side effects were correctly inferred. By accurately collecting these side effects, it is then possible to generate constraints that reflect the precise behaviour of each corresponding operation. Thus, it is of the utmost importance to determine whether the obtained results are reliable or not. To this end, we applied our solution to three common use cases, other than our running example: a simple *counter*; a *register* with increment, decrement, get and put operations; and a *calculator* that supports the four basic arithmetic operations (addition, subtraction, multiplication and division) as well as a set and a get method for its corresponding result. For all these use cases we have correctly identified all conflicting operations (for completeness, the output our compile-time analysis produces is presented in Appendix A).

Listing 5.1: Counter.java

```
1  class Counter {
2
3    private @Replicated int value;
4
5    public Counter(int initvalue) {
6      this.value = initvalue;
7    }
8
9    public int read() {
10     return this.value;
11   }
12
13   public void incr() {
14     this.value++;
15   }
16
17 }
```

5.1.1 Counter

This is a really simple use case where it is possible to *increment* and *read* a given *value* (see Listing 5.1). Since the *read* operation does not change the replicated state and since there is no system invariant that *incr* may violate (*value* is unbounded) they are both *locally permissible*. Our compile-time analysis does not find any conflict between these operations and thus, they both commute.

5.1.2 Register

In addition to the operations provided by the previous use case, a register also provides a *decrement* operation and a *put* operation being possible to set its value to a new desired value (see Listing 5.2). Similarly to the previous example, there are no restrictions regarding the replicated state. Therefore, no operation can introduce an incorrect state to the system. However, this time there are some operations that conflict with each other since the final result of performing both operations depends on the order in which they are performed, namely (*decr*,*put*), (*incr*,*put*) and (*put*,*put*).

5.1.3 Calculator

This use case represents a basic calculator where the four basic arithmetic operators are avaiable and where it is possible to *get*, *set* and *reset* its value (see Listing 5.3). The *get* operation does not conflict with any other because it has no side effects and thus, the replicated state would only be updated by the other operation (being the final value the same disregarding the order in which they were executed). Both *set* and *reset* operations conflict with the remaining ones. For the basic operators, other than (*div*,*plus*), (*div*,*minus*), (*mult*,*plus*) and (*mult*,*minus*), all operations commute.

Listing 5.2: Register.java

```
1  class Register {
2
3    private @Replicated int value;
4
5    public Register(int initvalue) {
6      put(initvalue);
7    }
8
9    public void put(int newvalue) {
10     this.value = newvalue;
11   }
12
13   public int get() {
14     return this.value;
15   }
16
17   public void incr() {
18     put(this.value+1);
19   }
20
21   public void decr() {
22     put(this.value-1);
23   }
24
25 }
```

5.1.4 Our running example

The bank account example that was considered throughout this document is a bit more complex than the considered use cases: it contains a system invariant (the non-negativity of its *balance*) and the *effects list* of its *transfer* operation has two parameters (*this* and *to*) instead of one as the previous operations. In the previous examples, if two operations were applied upon different objects, they would commute because they would not be manipulating the same replicated state. However, in this case, when we are evaluating a pair of methods that includes the *transfer* operation, for example (*transfer*,*withdraw*), even if both their corresponding side effects are parameterised with a different *this* parameter, it does not automatically mean that they commute.

As previously discussed, our static analysis correctly inferred, for a given pair of bank account operations, all parameterisations that cause them to conflict. However, this particular example contains a conditional effect and, as stated in the previous chapter, our support for conditional blocks simply unions the effects of its branches. A not so strict approach would be able to reduce the number of conflicts under these scenarios.

Figure 5.1 presents some statistics regarding our running example and each of the considered use cases: the number of methods each one of them comprises (where the constructor method was left out); the number of conflicting pairs of methods; and finally, the number of *locally permissible* methods that refers to the number of methods that may execute locally since they do not threaten any system invariants (e.g., the *deposit*

Listing 5.3: Calculator.java

```
class Calculator {

  private @Replicated float result = 0;

  public void set(float value) {
    this.result = value;
  }

  public void reset() {
    set(0);
  }

  public void plus(float value) {
    this.result += value;
  }

  public void minus(float value) {
    this.result -= value;
  }

  public void mult(float value) {
    this.result *= value;
  }

  public void div(float value) {
    this.result /= value;
  }

  public float get() {
    return this.result;
  }

}
```

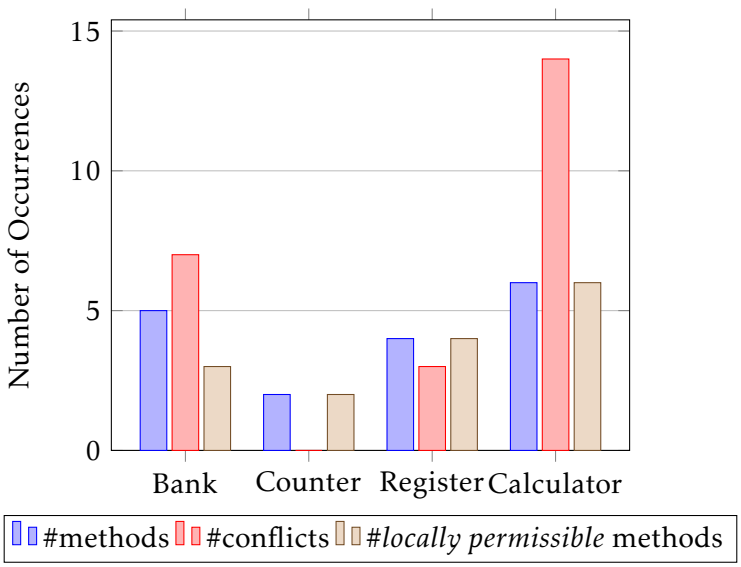


Figure 5.1: Statistics regarding the studied examples

operation). However, these metrics would be greatly affected if it could be possible to define that a particular method that manipulates a given replicated field needs to access its most recent version (not supported by our model). If in the previous examples it was possible to perform an up-to-date read operation concerning some replicated field, this operation would conflict with all system operations and could not be performed locally. Given that in all of the considered use cases there is no system invariants and none of the operations requires an up-to-date value, all their operations are *locally permissible*.

5.2 Performance

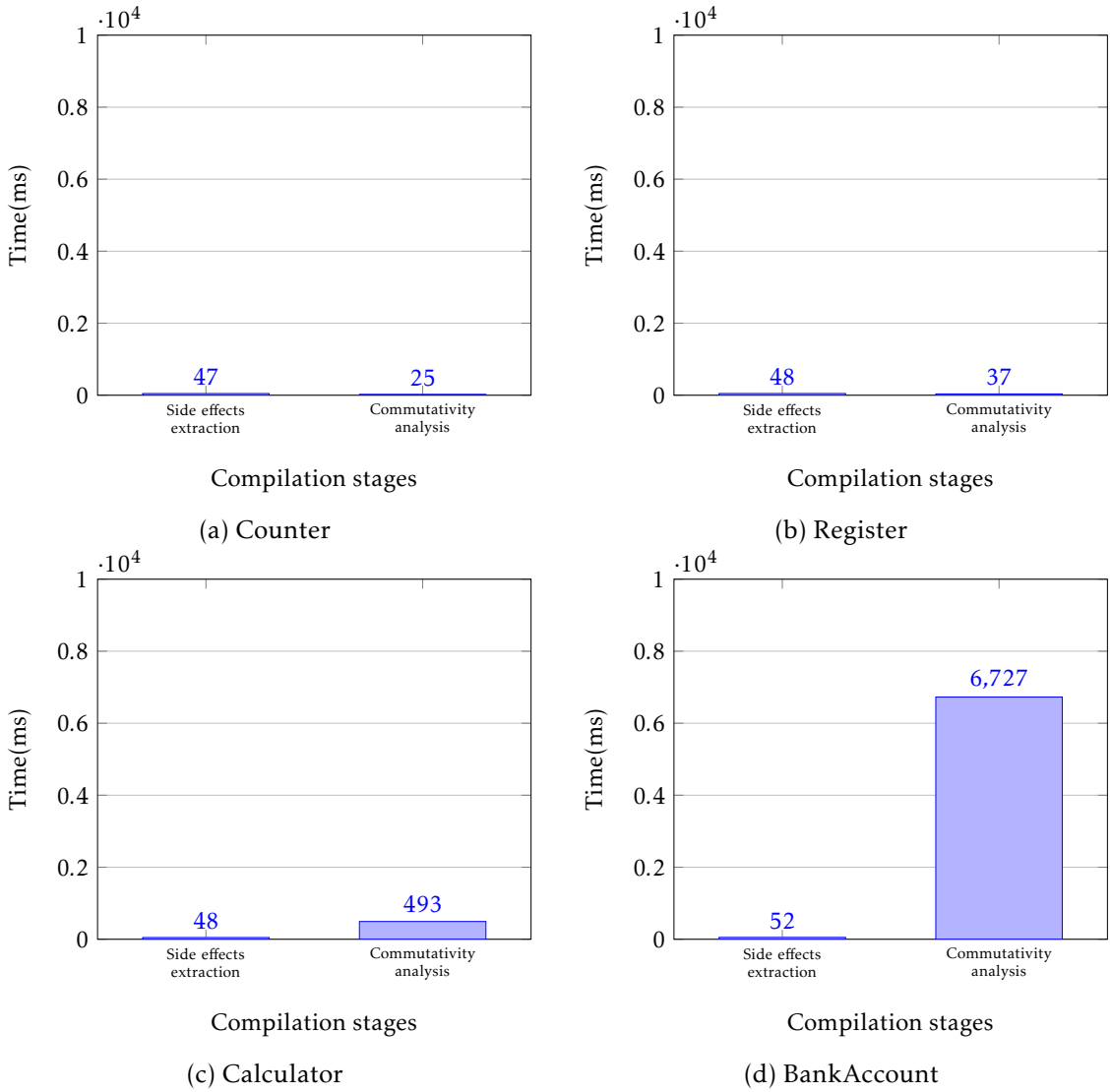
To assess the performance of our solution it is necessary to take into account the compile time overhead it incurs to perform all computations it requires. Moreover, it is also relevant to evaluate the additional compile time required when the number of replicated fields and the number of methods that manipulate the same replicated state vary. Incrementing these parameters translates into an increased interference between system operations which implies a longer time to perform all the commutativity analysis since it is necessary to consider all their *effects list* parameterisations that might cause two operations to conflict with each other.

5.2.1 Compilation Time Breakdown

From Figure 5.2, that shows the compile time overhead our solution incurs for the previously considered scenarios, we can infer that, as expected, it is the second compilation stage of our solution that takes longer. Interestingly, *despite* the calculator use case having more operations conflicting with each other, it is our running example (with half of the number of conflicts) that takes longer. Nevertheless, this is easily explained by the different number of parameterisations that have to be considered in both cases.

As depicted in Figure 5.3, in order to evaluate the compile time overhead our solution incurs, we varied the number of replicated fields each method manipulates (from 1 to 3) and the number of methods (from 1 to 15) where all interfere with each other, i.e., all handle the same set of replicated fields. The corresponding side effects of these operations only depend on one parameter (that is why even the worst case has incurred a lower overhead than our running example). According to these results, it is clearly noticeable that it is not the first stage of extracting effects that might jeopardise the feasibility of our solution. It shows that collecting the side effects of a given source code is relatively fast (the additional compile time it incurred did not exceed $65ms$), allowing us to identify that the bottleneck is in the second compilation stage - the commutativity analysis.

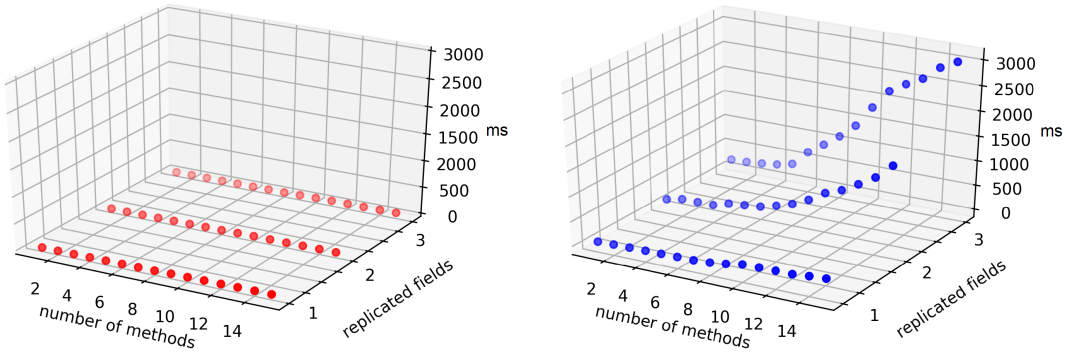
We can see that our solution is not the most advantageous when dealing with unlikely scenarios where each method of a given application manipulates and changes the state of many different replicated variables. The high overhead this phase incurs is easily justified



(a) Counter (b) Register

(c) Calculator (d) BankAccount

Figure 5.2: Compile time overhead our solution incurs



(a) Side effects extraction overhead (b) Commutativity analysis overhead

Figure 5.3: Compile time overhead breakdown

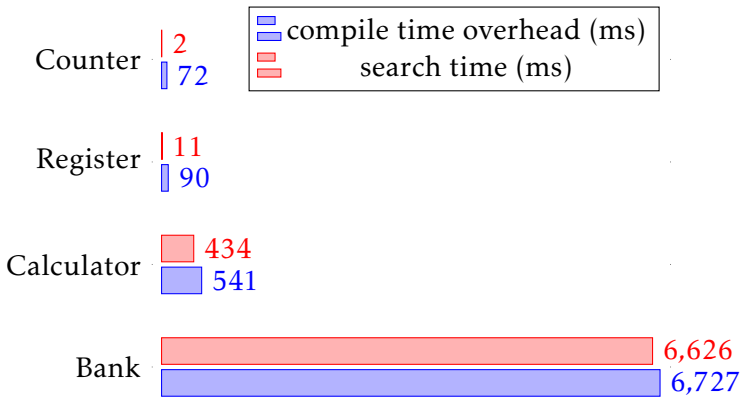


Figure 5.4: Search time compared to compile time overhead incurred

by the huge number of possible configurations it has to cover where for each pair of methods analysed, a constraint system that represents each one of these instance combinations must be conceived and solved (Figure 5.4 shows that, for all of the considered examples, the compile time overhead our solution incurs is mainly due to the time required to search for solutions). In the worst case scenario, where all methods interfere with each other and where their corresponding effects depend on multiple parameters, this approach is unlikely to achieve good results. However, for an ordinary replicated service containing the average number of methods, replicated variables, and methods that manipulate the same replicated variables (those that interfere with each other), our solution incurs an acceptable compile time overhead.

To sum up, so that the overhead that the presented solution incurs is really compensatory, the time it takes to examine the information that was generated by our analysis at runtime has to be minimal. Thus, checking this information must be as fast as possible otherwise our purpose is defeated and the latency that those replicated systems that were built following our approach will not observe as low latencies as expected.

Chapter 6

Conclusions

Taking into account the main challenges developers have to face nowadays when setting up a given replicated service, we presented our solution proposal based on an existing concurrency control model for shared memory systems (RC^3), extending it to cope with distributed memory systems (replicated systems). Although this work presents itself as a first step towards this goal, as a consequence of this new approach, the input the programmer is required to add to its original source code in order to specify its intended behaviour (and, hence, his reasoning) is greatly reduced when compared to the existing tools. Moreover, as a result of the compile-time analysis applied to the Java language that was developed, it is possible to automatically compute an operation pairwise commutativity analysis (i.e., identify which methods may conflict with others when concurrently executed) from that simple input - two keywords.

To demonstrate the feasibility of our solution we provided some results regarding its evaluation in terms of correctness and performance. It proved to be a feasible solution given the irrelevant compilation time it incurs for the average number of replicated variables that a particular service needs, as well as for the average number of methods that handle the same variable whose state is replicated that an ordinary service comprises. Evidently, for an unrealistic number of methods that contain multiple replicated parameters each it becomes almost impractical given the several possible instance configurations that have to be considered individually (build and solve a constraint system for each one of them).

6.1 Future Work

In the course of this work, a number of directions for future improvements have been identified. There are several open challenges at both model and implementation levels.

The next big steps to take are, of course, to extend the operation pairwise commutativity analysis to non-primitive types and to deal with non-private replicated fields. Covering non-primitive types will require more in-depth analyses as it will be necessary to create a new algebra over the heap and where it may not be possible to apply the solver as it is currently being applied. Dealing with non-private fields has a big implication that is: it will be no longer possible to do all this commutativity analysis at compile time. In the case of the Java language, only at the class loading and binding stage it might be possible to perform the commutativity analysis of the methods that manipulate this type of variables.

Since the bottleneck of compilation time is not in the early phase of side effects extraction but in the commutativity analysis phase, it may be necessary to rethink the search step that was applied. Adjusting this search method for finding solutions with a more elaborate and wise one would, perhaps, lead to better results and, consequently, to a higher scalability. Since all possible instance combinations have to be considered individually (which could threaten the solution scalability for some unrealistic scenarios) this search step must be as fast as possible when identifying conflicts between operations.

Furthermore, it is necessary to cover nodes referring to more complex constructors (such as loops) to cover a larger number of scenarios in order to corroborate the advantages our solution provides. And, finally, it is mandatory to develop some more complex examples which may introduce new kinds of problems so that the presented prototype can be improved accordingly.

Bibliography

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. "Causal memory: Definitions, implementation, and programming." In: *Distributed Computing* 9.1 (1995), pp. 37–49.

[2] F. Aleen and N. Clark. "Commutativity analysis for software parallelization: letting program transformations see the big picture." In: *ACM Sigplan Notices*. Vol. 44. 3. ACM. 2009, pp. 241–252.

[3] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting consistency back into eventual consistency." In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 6.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A critique of ANSI SQL isolation levels." In: *ACM SIGMOD Record*. Vol. 24. 2. ACM. 1995, pp. 1–10.

[5] P. A. Bernstein and N. Goodman. "Concurrency control in distributed database systems." In: *ACM Computing Surveys (CSUR)* 13.2 (1981), pp. 185–221.

[6] E. A. Brewer. "Towards robust distributed systems." In: *PODC*. Vol. 7. 2000.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 143–157.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's globally distributed database." In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[10] G. Dennis, R. Seater, D. Rayside, and D. Jackson. "Automating commutativity analysis at the design level." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM. 2004, pp. 165–174.

[11] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. "Commutativity race detection." In: *ACM SIGPLAN Notices*. Vol. 49. 6. Citeseer. 2014, pp. 305–315.

[12] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[13] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. "'cause i'm strong enough: Reasoning about consistency choices in distributed systems." In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 371–384.

[14] M. P. Herlihy and J. M. Wing. "Linearizability: A correctness condition for concurrent objects." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[15] F. Houshmand and M. Lesani. "Hamsaz: replication coordination analysis and synthesis." In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 74.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. Boston, MA, USA. 2010.

[17] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. "Consistency rationing in the cloud: pay only when it matters." In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 253–264.

[18] K. Kuchcinski and R. Szymanek. *JaCoP Library User's Guide*. http://jacopguide.osolpro.com/guideJaCoP.html. Accessed June-2019.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. "Providing high availability using lazy replication." In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 360–391.

[20] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[21] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[22] L. Lamport. "How to make a multiprocessor computer that correctly executes multiprocess progranm." In: *IEEE transactions on computers* 9 (1979), pp. 690–691.

[23] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making geo-replicated systems fast as possible, consistent when necessary." In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 265–278.

[24] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the choice of consistency levels in replicated systems." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 281–292.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 401–416.

[26] P. V. Mockapetris and K. J. Dunlap. "Development of the Domain Name System." In: *Computer Communication Review* 18.4 (1988), pp. 123–133.

[27] C. H. Papadimitriou. "The Serializability of Concurrent Database Updates." In: *Journal of the Assoclauon for Computing Machinery. Vol* 26.4 (1979), pp. 631–653.

[28] H. Paulino, D. Parreira, N. Delgado, A. Ravara, and A. Matos. "From atomic variables to data-centric concurrency control." In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM. 2016, pp. 1806–1811.

[29] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. "A commutative replicated data type for cooperative editing." In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 395–403.

[30] M. C. Rinard and P. C. Diniz. "Commutativity analysis: A new analysis technique for parallelizing compilers." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.6 (1997), pp. 942–991.

[31] Y. Saito and M. Shapiro. "Optimistic replication." In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81.

[32] E. Schurman and J. Brutlag. "Performance related changes and their user impact." In: *velocity web performance and operations conference*. 2009.

[33] A. W. Services. *Amazon S3 Introduces Cross-Region Replication*. https://aws.amazon.com/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/. 2015. Accessed: January 14, 2019.

[34] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[35] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis, et al. "Zeno: Eventually Consistent Byzantine-Fault Tolerance." In: *NSDI*. Vol. 9. 2009, pp. 169–184.

[36] K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan. "Declarative programming over eventually consistent data stores." In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 413–424.

[37] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional storage for geo-replicated systems." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 385–400.

[38] R. Van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *OSDI*. Vol. 4. 91–104. 2004.

[39] W. Vogels. "Eventually consistent." In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[40] YugaByte. *Isolation Levels | YugaByte DB Docs*. https://docs.yugabyte.com/latest/architecture/transactions/isolation-levels/. 2017. Accessed: February 22, 2019.

[41] N. Zaza and N. Nystrom. "Data-centric consistency policies: A programming model for distributed applications with tunable consistency." In: *First Workshop on Programming Models and Languages for Distributed Computing*. ACM. 2016, p. 3.

Appendix A - Produced Outputs

Operation	Parameters	Effects List
read	(this)	[ ]
incr	(this)	[=(this.value, +(this.value, 1))]

Table A.1: Side effects of each individual operation of the *counter* use case

incr(this)	*[ ]*
read(this)	*[ ]*

Table A.2: Conflicting operations for the *counter* use case

Operation	Parameters	Effects List
put	(this)	[=(this.value,newValue)]
get	(this)	[ ]
incr	(this)	[=(this.value,+(this.value,1))]
decr	(this)	[=(this.value,-(this.value,1))]

Table A.3: Side effects of each individual operation of the *register* use case

put(this)	*[put(this), incr(this), decr(this)]*
get(this)	*[ ]*
incr(this)	*[put(this)]*
decr(this)	*[put(this)]*

Table A.4: Conflicting operations for the *register* use case

Operation	Parameters	Effects List
set	(this)	[=(this.result,value)]
reset	(this)	[=(this.result,0)]
plus	(this)	[=(this.result,+(this.result,value))]
minus	(this)	[=(this.result,-(this.result,value))]
mult	(this)	[=(this.result,*(this.result,value))]
div	(this)	[=(this.result,/(this.result,value))]
get	(this)	[ ]

Table A.5: Side effects of each individual operation of the *calculator* use case

set(this)	*[set(this), reset(this), plus(this), minus(this), mult(this), div(this)]*
reset(this)	*[set(this), reset(this), plus(this), minus(this), mult(this), div(this)]*
plus(this)	*[set(this), reset(this), mult(this), div(this)]*
minus(this)	*[set(this), reset(this), mult(this), div(this)]*
mult(this)	*[set(this), reset(this), plus(this), minus(this)]*
div(this)	*[set(this), reset(this), plus(this), minus(this)]*
get(this)	*[ ]*

Table A.6: Conflicting operations for the *calculator* use case

ANNEX I

Annex 1 - JaCoP specifications

Variable type	JaCoP specification
FDVs	*IntVar(Store store, String name, int min, int max)* *BooleanVar(Store store, String name)*
FPVs	*FloatVar(Store store, String name, double min, double max)*

Table I.1: JaCoP specifications for defining variables

Floating point constraint	JaCoP specification
$\neg c$	*Not(c)*
$c_1 \vee c_2 \vee ... \vee c_n$	*Or(new PrimitiveConstraint[]{c_1,c_2,...,c_n})*
$c_1 \wedge c_2 \wedge ... \wedge c_n$	*And(new PrimitiveConstraint[]{c_1,c_2,...,c_n})*
if c_1 then c_2	*IfThen(c_1,c_2)*
if c_1 then c_2 else c_3	*IfThenElse(c_1,c_2,c_3)*

Table I.2: JaCoP specifications for defining logical and conditional constraints

Arithmetic constraint	**JaCoP specification**
$X = Const$	*XeqC(X, Const)*
$X = Y$	*XeqY(X, Y)*
$X \neq Const$	*XneqC(X, Const)*
$X \neq Y$	*XneqY(X, Y)*
$X > Const$	*XgtC(X, Const)*
$X > Y$	*XgtY(X, Y)*
$X \geq Const$	*XgteqC(X, Const)*
$X \geq Y$	*XgteqC(X, Y)*
$X < Const$	*XltC(X, Const)*
$X < Y$	*XltY(X, Y)*
$X \leq Const$	*XlteqC(X, Const)*
$X \leq Y$	*XlteqC(X, Y)*
$X \times Const = Z$	*XmulCeqZ(X, Const, Z)*
$X \times Y = Z$	*XmulYeqZ(X, Y, Z)*
$X \div Y = Z$	*XdivYeqZ(X, Y, Z)*
$X + Const = Z$	*XplusCeqZ(X, Const, Z)*
$X + Y = Z$	*XplusYeqZ(X, Y, Z)*
$X + Const \leq Z$	*XplusClteqZ(X, Const, Z)*
$X + Y \leq Z$	*XplusYlteqZ(X, Y, Z)*
$X + Y > Const$	*XplusYgtC(X, Y, Const)*
$\|X\| = Y$	*AbsXeqY(X, Y)*

Table I.3: JaCoP specifications for defining arithmetic constraints

Floating point constraint	**JaCoP specification**
$P = Const$	*PeqC(P, Const)*
$P = Q$	*PeqQ(P, Q)*
$P \neq Const$	*PneqC(P, Const)*
$P \neq Q$	*PneqQ(P, Q)*
$P > Const$	*PgtC(P, Const)*
$P > Q$	*PgtQ(P, Q)*
$P \geq Const$	*PgteqC(P, Const)*
$P \geq Q$	*PgteqQ(P, Q)*
$P < Const$	*PltC(P, Const)*
$P < Q$	*PltQ(P, Q)*
$P \leq Const$	*PlteqC(P, Const)*
$P \leq Q$	*PlteqQ(P, Q)*
$P + Const = R$	*PplusCeqR(P, Const, R)*
$P + Q = R$	*PplusQeqR(P, Q, R)*
$P - Const = R$	*PminusCeqR(P, Const, R)*
$P - Q = R$	*PminusQeqR(P, Q, R)*
$P \times Const = R$	*PmulCeqR(P, Const, R)*
$P \times Q = R$	*PmulQeqR(P, Q, R)*
$P \div Const = R$	*PdivCeqR(P, Const, R)*
$P \div Q = R$	*PdivQeqR(P, Q, R)*
$\|P\| = Q$	*AbsPeqQ(P, Q)*

Table I.4: JaCoP specifications for defining floating point constraints