NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RUBEN ALEXANDRE CORREIA VAZ

Bachelor in Computer Science and Engineering

IDENTIFYING OPERATION COMMUTATIVITY IN THE CONTEXT OF REPLICATED SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

IDENTIFYING OPERATION COMMUTATIVITY IN THE CONTEXT OF REPLICATED SYSTEMS

RUBEN ALEXANDRE CORREIA VAZ

Bachelor in Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Pedro Abílio Duarte de Medeiros
Associate Professor, NOVA University Lisbon

Rapporteur: Eduardo Resende Brandão Marques
Associate Professor, University of Porto

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2023

Identifying operation commutativity in the Context of Replicated Systems

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. Several systems resort to strong consistency to ensure high availability, although some systems are not allowed to weaken their consistency model.

Recognizing the tension between consistency and high availability, many systems allow multiple consistency levels to coexist. However, most of those systems require that the programmer explicitly specifies each operation's consistency level or declare the operation's invariants and side effects. Reasoning about the system's correctness becomes more challenging as the code scales. For instance, a single missing side effect identification may silently adulterate the application's behavior.

As part of the DeDuCe research project, this work aims to reduce the programmer's effort by only requiring the programmer to introduce a simple and intuitive input at data declaration. Following this approach, the reasoning is centralized, and all accesses to replicated data are identified automatically. By identifying which operations access to replicated data, these operations are analyzed for commutativity, and conflict relations are defined for each operations pair. Conflict-free operations will be refactored to a commutative version capable of executing without global coordination. In this context, this thesis focuses on extending compile-time commutativity analysis applied to the Java language able to compute operation pairwise commutativity from the input given at data declaration. In addition, to extend commutativity analysis to objects, we specify which are the conditions that guarantee commutativity for an operations pair.

We conclude our work by evaluating correctness and scalability. We show the correctness of the effects extraction and the produced outputs of each stage of the analysis. We analyze the time distribution in the inner stages of the analysis, and we seek to comprehend how the compile time scales when the size of the source code increases. Finally, we quantify the time required to determine if two method calls commute.

Keywords:
Distributed Systems, Replication, Commutativity Analysis, Concurrency control

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 e encaminhando os pedidos de acordo com a mesma. Existem alguns sistemas que são obrigados a enfraquecer a consistência de modo a garantirem alta disponibilidade para os seus utilizadores. Porém certos sistemas são demasiado sensíveis e não podem abdicar de consistência forte.

Tendo em conta as incompatibilidades entre propriedades como a consistência e a alta disponibilidade, alguns sistemas permitem que existam differentes niveis de consitência em simultaneo dentro dos mesmos. No entanto grande parte destes sistemas necessita de auxilo extra do programador. Assim sendo o programador necessita de especificar dependências/ordenações entre operações e invariantes a serem preservadas, ou até mesmo escolher o nível de consistência correto a ser atribuído para cada operação, o que é geralmente uma tarefa propensa a erros principalmente quando o programa escala.

No âmbito do projeto DeDuCe este trabalho visa a reduzir o esforço do programador exigindo uma contribuição simples e intuitiva por parte do mesmo ao nível da declaração de dados. Seguindo esta abordagem, o raciocínio é centralizado e todos os acessos aos dados replicados são identificados automaticamente. Após identificar quais operações acedem a dados replicados, estas operações são testadas de modo a verificar se comutam e são definidas relações para cada par de operações As operações que não tenham conflitos são refatorizadas numa versão comutativa capaz de executar sem coordenação. Neste contexto, o foco desta tese passa por estender a análise em tempo de compilação aplicada à linguagem Java capaz de computar a comutatividade entre operações a partir das anotações do programador. Para além de estender a análise de comutatividade a objetos, também serão especificadas as condições para que exista comutatividade nas operações.

Concluímos o trabalho avaliando a correção e a escalabilidade. É mostrada a correção dos efeitos extraídos e dos resultados produzidos por cada uma das fases da análise. É analisada a distribuição de tempo em cada uma das fases internas da solução, e é procurado perceber como escala o tempo de compilação quando o tamanho do código fonte é aumentado. Finalmente é avaliado o tempo necessário para determinar se duas

chamadas a métodos comutam.

Palavras-chave: Sistemas Distribuidos, Replicação, Análise de commutatividade, Controlo de concorrência

Contents

List of Figures

List of Tables

List of Listings

Glossary

SMT Satisfiability modulo theories generalize the SAT problem by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theorie [28]. 15

Acronyms

ACID Atomicity, Consistency, Isolation, Durability 9

1

Introduction

Nowadays, we live in a world where every second counts. According to Google in [10] users can detect latency differences in a few hundred milliseconds. Their research consisted of inserting slight delays (hundreds of milliseconds) in users' search responses, and they concluded that users tend to use less the search engine if the response times increased by a few hundred milliseconds. So every computing system tries to serve its clients in the shortest possible time period (minimum latency), which is lower bounded by the distance between the system and the clients.

When the system needs to process more clients or if clients are distributed across different regions, the simplest solution is to have more nodes. The system will have multiple nodes instead of a single node. All the nodes should keep the same data so that data will be replicated and clients can contact any replica (as nodes are replicated, we call them replicas), which allows distribution of the workload for all replicas. By having the system composed of a set of replicas, replicas can be placed in different regions to reduce latency. Data is replicated to boost performance and provide high availability and fault tolerance. Although data replication has its trade-offs, if any replica executes an operation and updates a value, that replica will have a different value from all the others, and there is a lack of consistency.

Consistency means that every replica has the same data view at a given time, irrespective of whether the client has updated the data [1]. When you request any node, you receive the same response, so from the outside, it looks like there is a single node performing all the operations. To ensure consistency, before a replica answers the client, the replica should propagate the client's request to all the other replicas. Thus every replica would keep the same data. However, making the client wait for all replicas to store a consistent state will decrease performance even more if replicas are distributed in different locations.

Another problem that emerges in large distributed systems and cannot be controlled by the systems is network partitions, which implies that some replicas cannot contact each other. Thus the system can also be partitioned. Therefore the system can only ensure that some replicas are consistent.

According to the CAP theorem [8, 18] it is not possible for a distributed system to simultaneously be (strongly) consistent , (highly) available and partition tolerant. Thus, systems can only support at most two of the following three properties at a given point in time:

(C)onsitency - 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.

Partition tolerance is a property that cannot be chosen. Therefore, systems have to choose between strong consistency or high availability.

Strong consistency models guarantee linearizability. Linearizability refers to serving requests concurrently where the order of the operations is set by the whole processes based on their time of occurrence. Linearizability requires a global synchronization clock called the global logical time. It ensures that a user always reads the latest committed write [1, 13].

Weak consistency [1, 14] models do not guarantee global synchronization among all the replicas. Thus, replicas can have different states, which can lead to losing data. For instance, a weaker consistency model, the Causal consistency model, ensures consistency between events with cause and consequence relationships. If a read operation results from multiple write operations on the replica, the read operation does not execute until the write operation has been terminated. This model guarantees that a client does not read two related write operations in the wrong order. In case the client has read the latest value, then it does not read the stale value. However, different replicas may observe concurrent writes in different orders, which can lead to data divergence among the replicas [1, 13].

Systems such as Amazon DynamoDB [14] or Cassandra [25] tend to relax the consistency property by resorting to a weaker consistency model, which allows these systems to provide low latency for client operations and achieve high scalability. There are, however, some systems which are critical and whose correctness requires strong consistency guarantees.

For instance, a banking service needs its data to be consistent for specific operations, such as a withdrawal. Otherwise, having divergent values in sensitivity data could lead to a catastrophic loss.

1.1 On the Combination Weak and Strong Consistency Models

Once it is impossible to weaken the consistency of some systems, a possible solution is to have methods that execute under a weak consistency model and, on the other hand, execute sensitive methods that require coordination under a strong consistency model.

Works, such as [5, 19, 21, 36], have been investigating solutions that allow different operations to execute under different consistency levels. Works such as [5, 19, 21], from invariant and operation's effect specification recurring to solvers are capable of detecting which operations are conflict-free and thus can execute without global coordination. However, the effects of these operations must be propagated among all the replicas in background to ensure state convergency. [5] has two different strategies for operations that may break the invariants(conflict) that can be chosen from the developer. They are executed under global coordination, or they execute, and in case of divergence, they are merged using merging rules. [19] presents a similar approach, but it resorts to token definitions to evaluate conflicts by defining cause-effect dependencies. It at least ensures causal consistency. Additionally, operations that do not require coordination require commutativity to ensure state convergence. On the other hand, [21] relies on the commutativity analysis to define the conflict operations. From commutativity analysis, it is possible to define conflict and dependence relations for each operation. Then, operations without conflicts or dependencies are free to execute under weak consistency. [36] presents a solution where developers define the consistency models they pretend to use so it is possible to have methods with different consistency levels.

Consider the example of coordinating operations over bank accounts replicated at several replicas where each account must have a non-negative *balance* (Fig. 1.1). From the perspective of each replica, given a sequence of incoming operations, it is fundamental to know which of these may execute locally, without coordination with the other replicas, and which ones have to execute in coordination with other replicas. Moreover, for operations that do not break invariants and hence may be executed locally, there is the second challenge of knowing if these may execute immediately or if they must wait for the conclusion of one or more operations presently coordinating for execution on a globally consistent state. In order to ensure that the final state is consistent and decide if the operation need to wait for the conclusion of some operation, commutativity can be used to evaluate the final state equivalence. If two operations commute, then the final state is equivalent (Section 2.1.2). If the evaluating operation commutes with all pending operations it does not require coordination with them.

Following the execution of Fig.1.1, a client calls a `deposit` operation. The `deposit` operation executes locally since the operation never breaks the invariant of having negative *balance*. After, a second client invokes a `transfer` operation, which is an operation that may break the invariant if the amount to transfer is greater than the *balance*. Thus `transfer` operation requires global coordination. While waiting for the response to the previous `transfer` request, the same client invokes a `deposit` operation. As discussed before, the `deposit` may execute locally, but can the `deposit` operation execute while there is a `transfer` operation waiting for coordination? Two operations are commutative if, after reordering their execution, the final result is the same. In our example `deposit` and `transfer` commute, since the *balance* is the same after executing the `deposit` after the `transfer` as executing the `transfer` after the `deposit`. Therefore a `deposit` operation

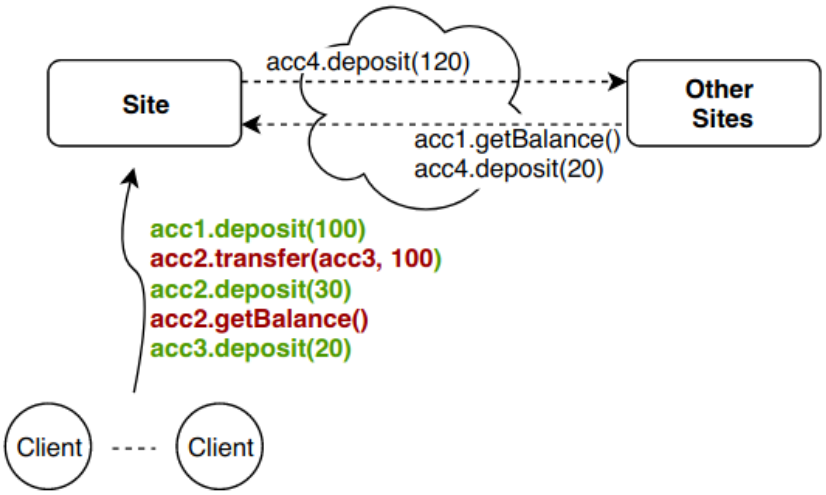

Figure 1.1: Bank with accounts replicated among multiple sites. Operations that may execute without global coordination are marked green, the remainder are marked red.

can execute locally before a `transfer` finishes the coordination.

1.2 Problem

The solutions proposed by the current state of the art [4, 5, 19, 21, 36] usually require that the programmer explicitly specifies each operation's consistency level or declare the operation's side effects. For instance, [5] says that it is required by the developer to define invariants and the effects for each operation. Additionally, the developer has to decide how to proceed with operations that possibly break the invariants. It allows executing them concurrently, defining conflict merge rules, or restricting concurrency to avoid invariant violation. The work presented in [19] requires the developer to specify the states, the program's initial state, invariants, a token declaration, and the operations' effects. In defining the operation effects, the developer must define if it is necessary to acquire a token to avoid conflicts and thus ensure coordination. [21] has similar requirements to [19], but it does not require the token, nor the token specification on operations effect. It requires only the states, the initial state, the invariants, and the operation effects. [36] requires the developer to define the level of consistency it desires for each operation precisely. Thus, it requires the developer to adapt its code and reason about the effects of his choices.

Reasoning about the system's correctness becomes more challenging as the code scales. For instance, a single missing or erroneous side effect identification or an incorrect choice of consistency level may silently adulterate the application's behavior.

1.2.1 The REPL Approach

The REPL work aims to reduce the programmer's effort by only requiring the programmer to introduce a simple and intuitive input at data declaration. Following this approach, the reasoning is centralized, and all accesses to replicated data are identified automatically. The REPL introduces the concept of expressing which fields are replicated among all the replicas simply by associate the field with @Repl annotation. There are three more properties defined in REPL which are the invariants, method conditions and consistent reads. Invariants and method conditions allow to define pre-conditions, one at field level and the other at the method level. Consistent reads allow the programmer to explicitly define that the method has to return the most recent value.

For instance, the Account class presented in Listing 1, is a REPL implementation of a simple account class and comprises a field, the *balance*, and five methods.

Field *balance* is associated with the concept of being replicated, expressed by the (`@Repl`) annotation.

The field is also associated with the `@Invariant` annotation which imposes restrictions on the field. In this example, the invariant requires that the field's value be greater or equal to zero.

The methods can also be associated with a method condition which is represented as an annotation (`@MethodCondition`), and takes a String to define the pre-condition expression. The pre-condition expression syntax is presented in Section 3.1.1.

The `deposit` method represents an operation of increasing the *balance* by the parameter, which is restricted by the method condition to be greater than zero. The `accrueInterest` represents the accrued interest, where the *balance* is multiplied by the parameter(interest rate) and summed to the original *balance*. The `getBalance` and the `getConsistent_Balance` return the account *balance*. The `withdraw` represents a withdrawal, which decreases the *balance* by the parameter, which is restricted by the method condition to be greater than zero. Finally, the `transfer` represents a transfer to another account. Thus, it decreases the *balance* by the *amount* parameter and increases the other account balance by the *amount* parameter, which is also restricted by the method condition to be greater than zero.

It is possible to observe that methods such as the `deposit`, the `accrueInterest`, or the `getBalance` never break the invariant associated with the *balance* field. This only happens because in the methods that alter the *balance* field, the only possible outcome is to increase the *balance* value, and thus, if the *balance* is greater or equal to zero when the *balance* is increased, it cannot be lower than it was. Thus, these methods never break the invariant, and they would be allowed to execute without global synchronization and, therefore, rely on a weaker consistent model.

The problem arises when there exist methods that can break the invariant, such as the `withdraw` or the `transfer`, which decrease the *balance* value and thus can make the

```java
public class Account {
    @Repl @Invariant("balance >= 0") private int balance;

    @MethodCondition(expression = "param0 > 0")
    public void deposit(int amount) {
        if (amount > 0)
        this.balance += amount;
    }

    @MethodCondition(expression = "param0 > 0")
    public void withdraw(int amount) {
        this.balance -= amount;
    }

    @MethodCondition(expression = "param1 > 0")
    public void transfer(Account to, int amount) {
            withdraw(amount);
            to.deposit(amount);
    }

    @MethodCondition(expression = "param0 > 0")
    public void accrueInterest(int interestRate) {
        if (interestRate > 0)
        this.balance = this.balance + this.balance * interestRate;
    }

    public int getBalance() {
        return this.balance;
    }

    @ConsistentRead
    public int getConsistent_Balance() {
        return this.balance;
    }

}
```

Listing 1: Account class REPL-Java

balance value lower than zero. Thus, executing these methods without global synchronization could lead to an invariant break. In a sensitive system such as a bank, letting its clients have their balance lower than allowed could represent a catastrophic event with a loss that the system could not recover.

This thesis aims to analyze the source code, check if two method calls commute and under which circumstances, and hence both methods can execute in parallel.

Several systems (introduced in Section 2.2) address different types of commutativity

analysis, but none of them can be directly applied to our work.

1.3 Proposal

We propose to create a solution that starts by analyzing the source code and extracting the side effects and invariants that affect replicated fields for each method. Additionally, it is also necessary to extract the method pre-conditions to ensure they are not broken. After the side effect extraction, we evaluate if two method calls commute using the Servois [6](described in Section2.2) which relies on an SMT-Solver. To evaluate if the method calls commute, we resort to the method's side effects, invariants, and pre-conditions, which have been extracted before. When evaluating if two method calls commute, we verify if two method calls do commute, and we define the conditions(parameters values or fields' states) for them to commute. Finally, from the result obtained from the commutativity analysis, we create a map of functions for each method call pair with the correspondent commutativity conditions. The solution output is the function's map with the commutativity conditions, which can be used at runtime to determine if method calls require strong consistency or can execute without global coordination.

1.4 Contributions

Our contributions for this thesis are:

- Algorithm to extract side effects from methods by analyzing the source code and identify if two method calls commute and which are the requested conditions for two method calls commute;
- Algorithm implementation in the Repl-Java context;
- An experimental evaluation of the proposed solution that shows its correctness and scalability.

1.5 Document Outline

The remainder of this document is structured as follows:

Chapter 2 introduces the fundamental background and related work. It starts by defining crucial concepts and properties related to commutativity. It presents the different types of commutativity analysis and a brief description and introduction to SMT and SMT-solver, where the existing SMT theories and the most common solvers are presented. Finally, the related work and a discussion comparing the different solutions is presented.

Chapter 3 briefly describes the developed solution and presents some considerations that a developer who extends the program must have.

Chapter 4 describes in detail the implementation of each phase of the developed solution, starting with the side effects extraction, passing by the commutativity analysis, and finishing with transforming the commutativity conditions into Java functions.

Chapter 5 presents the experimental results. The evaluation of classes scalability and the conditions verification times are also evaluated.

Chapter 6 summarises the conclusions related to the implementation of this project and the elaboration of this thesis and describes possible changes and extensions that may be added to the solution.

2

Background and Related Work

After the introduction of the goal of this thesis in the previous chapter, this chapter aims to cover the basic notions needed to understand the concept of the system. Section 2.1 briefly describes fundamental properties and main concepts related to commutativity analysis, followed by an analysis of the existing systems in Section 2.2.

2.1 Background

2.1.1 Atomicity

Atomicity is one of the ACID properties from Database systems that is applied on transactions. The key idea is, inside a transaction, all operations have their intended effects, or none have.

Weihl [38] states that transactions are atomic if their execution appears to be serializable and recoverable, i.e., a system's history is atomic if the committed transactions in the history can be executed in some serial order and have the same effect.

REPL builds upon RC3 [32] that defines each method that operates over an atomically annotated value (in our case, a replicated annotated value) as a unit of work that must execute in mutual exclusion, which matches the definition mentioned in [38] since operations will behave like they were executed sequentially.

This notion of atomicity present in Repl will guarantee that in a parallel environment, all operations will not have unpredictable behavior, and the programmer will not need to worry about concurrent accesses.

2.1.2 Commutativity

The *commutativity* property is most known as a mathematical property where the order of operations inside an equation may change, being that the final result is the same. The most common example is the sum of two numbers:

$$x + y = y + x$$

In predicate logic, the same principle is applied to the conjunction and disjunction according to the commutative laws:

$$A \wedge B \Leftrightarrow B \wedge A$$

$$A \vee B \Leftrightarrow B \vee A$$

This same principle may be applied to computer programs. Given an object, a pair of deterministic operations (α, β) and some initial sate S, then:

$$S_{\alpha\beta} = S_{\beta\alpha} \implies \alpha \bowtie \beta$$

where $S_{\alpha\beta}$ denotes the resulting state from the execution of operation α over state S followed by β, $S_{\beta\alpha}$ stands for the resulting state from the execution of β followed by α and $\alpha \bowtie \beta$ indicates that α and β commute.

As we will detail in Section 2.1.5, different types of commutativity analysis exist, each with its own properties and methodologies. For instance, Rinard *et al.* [35] have a very restricted approach and only consider that operations commute if, after their reordering, the final result is the same (objects are equal in memory) and the execution flow is also the same, i.e., the multiset of methods invoked during both executions has to be the same, which includes having the same parameters. Others [39, 15, 2, 23] have a more relaxed approach and argue that the definition of an equivalence between two final states is enough. Concretely, in [15], programmers must specify equivalence tests using an auxiliary predicate that defines the equivalence of states in terms of their components. A possible equivalence test for a queue is to define that two queues are equivalent if they have the same elements, even if those elements are not in the same order. In [39, 2] operations commute if the returned output is exactly the same for each operation after reordering the execution. P. Wu *et al.*[39] refer that two methods may commute just by changing the return type. The work presented in [23] applies to the context of *algorithmic skeletons* [12], being that only variables visible outside a skeleton's scope are used to check commutativity.

To reach a state of equivalence, it is necessary to ensure that both operations can occur, i.e., pre-and post-conditions should not be violated in the new execution order, so F. Wang et al. refer in [15] that, given two atomic operations, commutativity is achieved if following two properties are met:

Diamond Connectivity - α and β are diamond connect if both occur from the initial state, α can occur after β and β can occur after α. If two operations are diamond-connected, the execution of one operation cannot prevent the execution of the other, so permuting those two operations create a valid execution (Figure 2.1 (i) and (ii)). Contrarily, if two operations are not diamond connected, i.e., one of the sequences is not possible from the initial state (Figure 2.1 (iii)), the execution of that sequence should result in errors due to the imposed order over those operations.

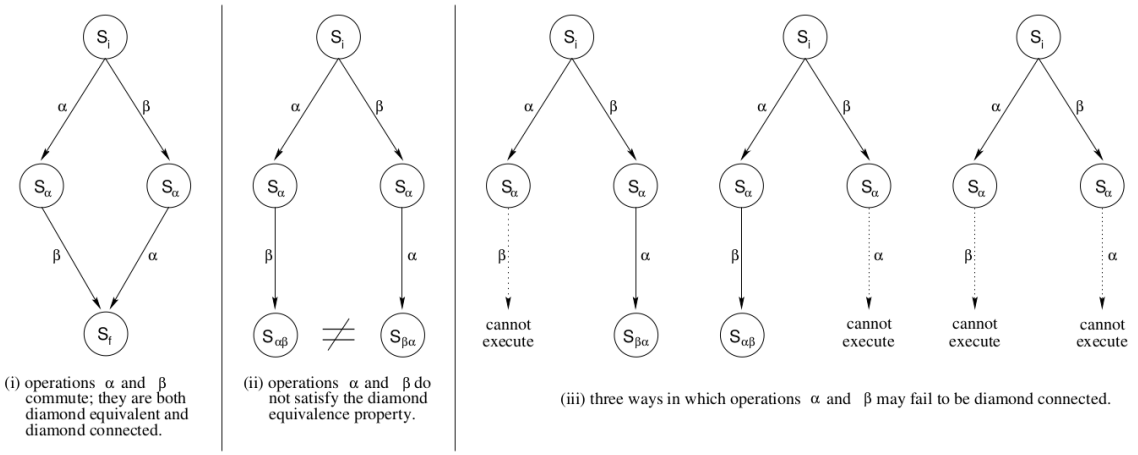

Figure 2.1: Commutativity properties (taken from [15])

Diamond Equivalence - Two operations α and β are diamond-equivalent if they are diamond-connected and the final state of $\alpha\beta$ and $\beta\alpha$ are equivalent (Figure 2.1 (i)). Thus, if two operations are executed simultaneously by different users, it does not matter which is executed first because both execution orders will converge for the same result. On the other hand, if two operations are not diamond-equivalent and are executed in different orders, the result may be unpredictable.

F. Wang *et al.* express operations as a constraint parameterized by its pre-state S and its post-state S', explicitly separating the pre- and post-conditions:

$$Op(S, S') = Pre(S) \wedge Post(S, S')$$

F. Wang *et al.* also mention that pre-conditions represent guards and not disclaimers, so any invoked operation in a state that violates a pre-condition will be discarded. Thus for operations α and β, we have:

$$\alpha(S, S') = Pre_{\alpha}(S) \wedge Post_{\alpha}(S, S')$$

$$\beta(S, S') = Pre_{\beta}(S) \wedge Post_{\beta}(S, S')$$

α and β are diamond-connected if for all states S_i, S_α, S_β:

$$\alpha(S_i, S_\alpha) \wedge \beta(S_i, S_\beta) \implies Pre_{\alpha}(S_\beta) \wedge Pre_{\beta}(S_\alpha)$$

α and β are diamond-equivalent if for all states S_i, S_α, S_β, $S_{\alpha\beta}$, $S_{\beta\alpha}$:

$$\alpha(S_i, S_\alpha), \beta(S_\alpha, S_{\alpha\beta}) \wedge \beta(S_i, S_\beta), \alpha(S_\beta, S_{\beta\alpha}) \implies S_{\alpha\beta} = S_{\beta\alpha}$$

2.1.3 Commutativity Categories

In [39], P. Wu *et al.* define commutativity analysis according to a relationship between different notions of conflict applied to object databases. In such database systems, locks are used on each instance of a class, and a few use locks at the attribute level within a class instance. Given this context, P. Wu *et al.* discuss a locking strategy under commutativity.

Read/Read pairs are conflict-free operation pairs under read/write locking and commutativity. If two operations read the same variables, read operations do not modify any variable and hence cannot change the behavior of the other operation. Consequently, a read/read pair of operations is always commutative.

Different field pairs is applied to write/write or write/read operations that access different attributes. Read/read pairs over different fields are placed on the *Read/Read pairs* category.

Write operations that write on different attributes always commute since only one of the operations will change each attribute. If operation `inc_A` increases the variable a and the operation `inc_B` increases the variable b. If at an initial state, $a = 0$ and $b = 5$, the execution of `inc_A` followed by `inc_B` will produce, $a = 1$ and $b = 6$, and the execution of `inc_B` followed by `inc_A` will also produce $a = 1$ and $b = 6$.

Semantic commutative pairs are two operations acting on the same field of the same instance, with at least one of them being a write operation, but may commute, depending on their semantics. One example is a pair of bank deposit operations over the same account. If the initial balance is 0, and the two deposit operations are executed, `deposit(10)` `deposit(20)`, the final balance will be 30; if the deposit operations are executed in a different order, `deposit(20)` `deposit(10)`, the final balance is also 30.

Even if a deposit operation commutes with itself, two deposit operations cannot execute in parallel. A parallel execution of two deposit operations could lead to an undesired result; take the following example. When `deposit(10)`and `deposit(20)` are executed in parallel, if both operations read *balance* simultaneously, *balance* is equal in both operations and then both write. In the end, the final *balance* will be only the result of the operation that writes last. So even if operations commute, operations must be atomic.

2.1.4 Forward and Backward Commutativity

The concept of *forward and backward commutativity* is presented in [38]. However, left and right movers, a similar notion is also presented in [6]. In the latter, this concept is an asymmetric version of commutativity where a method commutes in one direction and not in the other. A right-mover is an action α_1 that moves to the right (forward) of action α_2, denoted as $\alpha_1 \triangleright \alpha_2$. Left-movers (moves backward) can be defined as right-movers, but with arguments swapped and are denoted as $\alpha_2 \triangleright \alpha_1$.

If a pair of methods is simultaneous right-mover and left-mover, that pair is commutative. The pair is non-commutative if it is neither a right-mover nor a left-mover.

W. E. Weihl. [38] presents the same idea, but instead of applying this notion to a pair of actions, it is applied to a pair of sequences of actions. Thus if M is a state machine and R and S are two sequences of transitions of M, R and S commute forward if for every state s in which R and S are defined $R(S(s)) = S(R(s)) \neq \perp$. Backward commutativity is

method1		*method2*	*condition expression*
decrement	$\bowtie$	decrement	$true$
increment	$\triangleright$	decrement	$\neg(0 = counter)$
decrement	$\triangleright$	increment	$true$
decrement	$\bowtie$	reset	$false$
decrement	$\bowtie$	zero	$\neg(1 = counter)$
increment	$\bowtie$	increment	$true$
increment	$\bowtie$	reset	$false$
increment	$\bowtie$	zero	$\neg(0 = counter)$
reset	$\bowtie$	reset	$true$
reset	$\bowtie$	zero	$0 = counter$
zero	$\bowtie$	zero	$true$

Table 2.1: Counter commutativity analysis (taken from [6])

similar to forward commutativity, but instead of moving operations forward, they are moved backward.

Table 2.1 presents the application of forward and backward commutativity to a counter class restricted to non-negative values and features four methods: `increment`, `decrement`, `reset`, and `zero` test. If we look at the execution of an `increment` followed by a `decrement`, `increment` is only a right-mover if the initial value is bigger than zero. On the other hand, if `decrement` is followed by `increment`, `decrement` is always a right-mover, and thus `increment` is always a left-mover. Given that the preconditions (right column) differ depending on whether `increment` is a left or a right mover, then the operation pair (`increment`, `decrement`) does not commute.

2.1.5 Types of Commutativity

On the authority of [23], commutativity can be classified into three categories: Separability-Based Commutativity, Output-Based Commutativity, and Liveness-Based Commutativity.

Separability-Based Commutativity - considers that methods can be decomposed into two parts. The "*object section*", where the object is accessed and can be modified, and the "*invocation section*", where other operations can be invoked. If methods cannot be decomposed in these two sections, then methods cannot be evaluated for commutativity.

In [35], the foundations for commutativity based on separable methods are defined as **Instance Variables** and **Invoked Operations**.

Instance Variables - The value of each instance of receiver objects A and B must be equal if executing object section A and then object section B or executing object section B and then object section A.

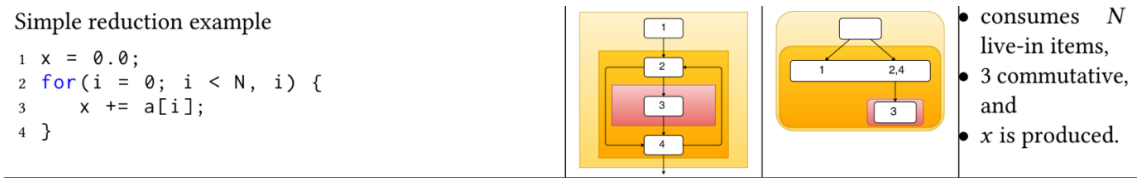

Figure 2.2: Liveness-based Commutativity example (from [23])

Invoked Operations - The multiset of operations directly invoked by A and B must be the same if executing A followed B and executing B followed A. Two operations are the same if they execute the same method with the same parameters.

The applicability of separability-based commutativity to real-world applications is very limited [23] since it requires an object-based program and imposes several major restrictions, such as no global variables and no inheritance, among others.

Output-Based Commutativity - is proposed for individual functions (it is limited to the repeated, possibly commutative invocations of a single function), and checks the output of the function at the time it is used, i.e., it will only verify if operation commutes when some function exposes the output of the affected variables. A candidate function is symbolically executed in two different call orders to create an abstract representation of the result for each order. The result is then used as input to all functions that could potentially read the candidate function. These "reader"functions are symbolically executed, and if the outputs of these reader functions are identical, then the candidate function commutes.

Suppose two users insert values in a set, and the return of inserting is a Boolean. In that case, if both users insert the same value, although the set's final state is the same, each user will receive a different result if the order of execution changes. If the operation issued by *userA* is executed first, then *userA* receives true, and *userB* receives false. If the order of execution is changed, *userB* receives true, and *userA* receives false. The insert operation does not commute because for the same user, if the execution order is different, the output is also different.

Liveness-Based Commutativity - is introduced as a new definition in [23] and combines the best from separability-based commutativity, "incorporate the power to reason about any two(or more) regions of the code"and output-based commutativity where output is considered, but only evaluates values that will or might be used later.

One of the examples presented is a simple reduction function (Fig. 2.2) where initially code is grouped by blocks which are called "skeletons". Then the operation inside the loop (which is a block) is commutative because it only consumes a live-in value (a value that is only accessed inside the block, a[i]) and the live-out value (a value that is accessed outside the block), which is x has the same value even if the loop order changes.

2.1.6 Analysing Commutativity

In this subsection, we will discuss some techniques used to verify the correctness of programs. In our system, it may be used to verify operations commutativity. We should note that some of these techniques may be used together.

2.1.6.1 Static vs Dynamic Analysis

Static analysis tests and evaluates an application by examining the code without executing the application [16]. The static analysis aims to confirm some aspect of the program regardless of the input. It examines all possible execution paths and variable values, not just those invoked during execution. A common use of static analysis is inside compilers, so we can refer to static analysis as compile-time analysis.

Dynamic analysis tests and evaluates an application during runtime, so it can also be classified as runtime analysis. It involves running the program and comparing the returned outputs with the expected outputs [27].

The most common example of dynamic analysis is testing. By testing a program, it is possible to observe the program's behavior when executed with a wide variety of expected inputs, thereby verifying if the program works as intended. One big problem of testing is that, in most cases, it is impossible to test all possible scenarios, and concrete executions can only under-approximate the analysis to a property of interest.

2.1.6.2 Symbolic Execution

Symbolic execution is a solution that neither is static neither is dynamic. It executes before runtime but is not static because it does not ignore the inputs. Symbolic execution is a way of executing a program abstractly so that one abstract execution covers multiple possible inputs of the program and simultaneously explores multiple paths. Symbolic execution is widely used in the context of commutativity analysis, in [35] presents that the input in the method used by a symbolic executor can be an expression instead of a value. At the end of the symbolic execution, each variable can be constrained in various ways or be a concrete value. Then, with all the variables in their respective constraints, it is possible to ask the solver if the constraints can be satisfied.

2.1.6.3 SMT Solvers

Constraint solvers are decision procedures for problems expressed in logical formulas. One of those problems is the Boolean satisfiability problem (SAT), which aims to determine if an interpretation of the symbols of a formula exists that makes it true. As an NP-complete problem, there is no known algorithm to determine the satisfiability of a Boolean formula in polynomial time. SMT generalizes the SAT problem to more complex formulas involving real numbers, integers, and data structures [28].

An SMT solver is a tool that aims to solve the SMT problem. Solvers have been used as a building block for various applications such as program analysis, program verification, and software testing. Solvers have a little issue; if the solver is asked to solve if two states are the same, it will look for a single initial state, although the main goal is to check if two final states are equal for any initial state. So by turning the question around, i.e., we ask the solver to find some initial state where the final state is different. If the solver fails to return an example, it is assumed that both operations commute. So instead of asking to solve:

$$\forall S(S_{\alpha\beta} = S_{\beta\alpha} \implies \alpha \bowtie \beta)$$

We should ask:

$$\exists S(S_{\alpha\beta} \neq S_{\beta\alpha} \implies \alpha \not\bowtie \beta)$$

2.1.6.4 SMT-Lib and most famous SMT-Solvers

Z3 and CVC4 are the most used SMT-Solvers. They can prove and build a model where the inputted formula is satisfied, they support the same theories and both are developed in C++. However, only Z3 can generate proofs for unsatisfiable formulas and extract unsatisfiable cores.

According to [7], a theory is a set of sentences, and it has been said that a formula Φ is satisfiable modulo a theory T, if $T \cup \Phi$ is satisfiable, meaning there exists a model M that satisfies Φ under the theory T, and the defined theories are:

Uninterpreted functions with equality is denoted as QF_UF: quantifier-free uninterpreted functions with equality. An uninterpreted function is a function with a name and arity; however, it has no interpretation, i.e., functions are just an abstraction, and the function's behavior is ignored, *f(x),g(f(x)),f(x,y)* are some examples of uninterpreted functions. The theory permits, equalities (=), unequalities (! =) and boolean connectives ($\wedge, \vee, ...$).

Linear arithmetic is denoted as LIA, LRA, QF_LIA, and QF_LRA, meaning quantified or quantifier-free linear integer arithmetic or linear real arithmetic. The theory permits sums (+) and subtraction (−). Additionally, it allows the use of relational symbols ($<, \leq, \not\leq, ...$) to form the predicates.

Diference arithmetic is denoted as QF_IDL and QF_RDL, which symbolize quantifier-free integer difference logic and quantifier-free real difference logic. It is part of the Linear Arithmetic Theory and restricts inequalities to be in the form $x - y \leqslant c$, where x and y are variables and c is a constant.

Non-linear arithmetic is denoted as NIA, NRA, QF_NIA, and QF_NRA, which stands for quantified or quantifier-free non-linear integer arithmetic and non-linear real

arithmetic. The decision procedures over reals use algorithms from computer algebra. According to [20], deciding satisfiability for this theory problem is undecidable, meaning no algorithm can solve every instance of this problem.

Bit-vectors is denoted as QF_BV: quantifier-free bit-vectors. It represents every number as a fixed-size sequence of bits. In addition to standard arithmetic operations, it also allows mixing bit-wise operations, such as NOT, AND, OR, XOR, and bit shifts.

Arrays is denoted as QF_AX: quantifier-free arrays with extensions. It defines the usage of arrays, which have two functions, *write(v,i,a)*, which means that the value v is written in the index i on the array a. The second is *read(a,i)*, which stands for reading the index i from the array a. The most common approach to deal with the theory of arrays is to use a reduction to the theory of uninterpreted functions (QF_UF).

Quantified Theories defines the usage of quantifiers to form predicates, such as $\exists$ or $\forall$, but with it's usage the problem of deciding satisfiability becomes significantly more difficult .

Furthermore, it is possible to define the **Theory Combination**, which consists in handling the previous theories combined efficiently. There were developed some methods trying to solve the problem of a combination of theories for SMT, some of them are the **Nelson-Oppen** [11], **Delayed Theory** [11], and **Model-based Theory** [29].

According to [20] while competing in SMT-COMP, which is a competition to evaluate which of the solvers have better benchmarks for each theory, Z3 has been dominating the competition over the years for almost all theories and also refers that Z3 could solve 15 benchmarks that any other solver could not. Both can be used with an API, but Z3 wins this battle once CVC4 only has an API for C++ and Python; Z3 has an API for C, C++, .NET, Python, Java, and OCaml, but we should remember that smt-lib is the standard input language for most of the SMT-Solvers including Z3 and CVC4, where both support version 2.

SMT-Lib was developed to facilitate the research and development of SMT because a common standard and a library of benchmarks would facilitate the evaluation and the comparison of SMT-Solvers. The language is command-based. It includes commands for setting various solver parameters, declaring new symbols/functions, asserting formulas, checking the satisfiability of the given assertions, inquiring about models of satisfiable sets, and printing various diagnostics.

As described in [7] the main features of the SMT-lib standard are:

- a language for writing terms and formulas of first-order logic;
- a language for specifying background theories and fixing a standard vocabulary of sort, function, and predicate symbols for them;

- a language for specifying logics, suitably restricted classes of formulas to be checked for satisfiability concerning a specific background theory;

- a command language for interacting with SMT solvers that allows asserting and retracting formulas, querying about their satisfiability, and examining their models or their unsatisfiability proofs.

2.1.6.5 Random Interpretation

Random interpretation is a hybrid technique introduced by F. Aleen and N. Clark. [2], which combines the simplicity of testing with random inputs while enabling the compiler to achieve correctness very close to 100%.

The first step in the algorithm is to generate random values. These values will be mapped to declared variables and operation arguments, building an initial state that will be used for testing.

Random interpretation requires that all possible control flow paths are executed, so in the case of a branch, the previous state is cloned, and variables that will be evaluated to decide the flow of the branch are changed in a way to follow a different path.

If the equation cannot be solved, e.g., if the branch condition is `if (1 == 0)`, then that control flow path is discarded. After the execution of code under the branch block is finished, the different flow states are merged into one.

The merge operation creates a weighted average of the values computed on the different paths of execution, using the equation $\phi_w(v_1, v_2) = wv_1 + (1 - w)v_2$, where v_1 and v_2 are the values being merged, and w is a randomly generated weight. Some randomly generated inputs and weights may lead to incorrect results. For example, a weight of 1 or 0 will effectively destroy the state from one execution path. If weights are selected from the set of integers on a 64-bit machine, this bounds the chance of error to at most $\frac{\#joins}{2^{64}}$, which in practice is a negligible probability.

If the states were not merged, the size of the state would exponentially explode, which is the main problem with most equivalence checkers.

2.2 Related Work

Now we will discuss some of the existing systems that analyse commutativity or use commutativity for a final purpose. We can split these systems according to their main purpose, some are auxiliary tools, which the main purpose is to verify if two operations are commutative or in which conditions operations commute [15, 6], or alert . The others [35, 39, 2, 33, 21, 26] will verify if operations commute to parallelize their programs to gain performance.

2.2.1 Semantic Foundations of Commutativity Analysis [35]

In addition to the introduction of commutativity based on separable methods (Section 2.1.5), Rinard *et al.* developed a static analysis algorithm to verify operations commutativity under separability-based type conditions. The presented system is applied to Object-Oriented Programming, where the primary purpose for commutativity analysis is computation parallelization, which can be achieved if two methods commute. It is described in the context of separable methods, i.e., methods that can be separated in the object section (changes or reads the object state) and invocation section (invokes another methods/functions), methods that are not separable will not be available for commutativity analysis. Thus will always be considered non-commutative.

The algorithm is based on a symbolic execution that executes methods with expressions instead of values. It keeps a set of bindings that map variables to the expressions that indicate their values. To test if the executions of two methods commute, the compiler first uses symbolic execution to extract expressions representing the new values of the instance variables and the multiset of invoked operations for both execution orders. Expressions are simplified and compared. If they express the same value, then both methods commute.

2.2.2 An Empirical Study of Commutativity in Application Code [39]

After introducing the commutativity categories (Section 2.1.3), P. Wu *et al.* extend their study not only to identify the operations that commute but also the reasons for their commutativity according to the commutativity categories. This system is applied to Object-Oriented Programming and aims to find which method calls can execute concurrently. It is presented in the context of methods calls within the same class, so it will evaluate if methods from the same class commute. These are simple methods that change class variables. Thus they can be defined as separable methods, and we can infer that this system belongs to the separability-based type.

To verify commutativity between methods, they used the introduced technique of commutativity categories. For semantic-based commutativity, authors do not describe the process of analyzing commutativity under those terms. The authors also mention that only methods from the same class are evaluated since "operations of a particular class will mostly have reads or writes that are relevant only to that class".

A matrix $N \times N$ is drawn for each class, where N is the number of methods in the class. Each operation conflict pair within the matrix is analyzed individually to determine whether they commute as read/read pairs, different field pairs, semantic commutative pairs, or do not commute. Each matrix entry will contain the commutativity category that makes operations commutative or "n" for the non-commutative operations pair.

In the end, the authors only discuss the concurrency potential for each class and present a commutativity analysis summary where it is possible to observe that only a few

commutative pairs belong to semantic commutativity, and the most commutative pairs come from different field pairs.

2.2.3 Automating Commutativity Analysis at the Design Level [15]

This system is an auxiliary tool that aims to help program designers to preview operations that may execute in parallel. The system is expressed in OCL, the constraint language of UML, which permits to analyze commutativity on objects. This work is presented for a treatment unity where it was necessary to manage the patients waiting queue. This work introduces commutativity analysis as constraint solving, where the designer specifies the constraints, operation effects, and assertions. Commutativity is achieved if the constraints and assertions are fulfilled. To verify these conditions, it is used a constraint solver.

It used the Alloy Analyzer, a first-order relational logic solver MIT Software Design Group developed for this project. Alloy Analyzer searches for a scenario where constraints or assertions fail. The Alloy Analyzer does not symbolically prove whether or not instances of non-commutativity exist. It exhaustively searches the entire state space of scenarios within specified bounds. It analyzes millions of scenarios, but not all. If it fails to find a counterexample, it does not constitute proof that the claim (assertions and constraints) is valid. However, the reporting of a counterexample implies that it is invalid.

To verify operations commutativity, it is necessary to ensure that operations meet the diamond properties (Section 2.1.2). To check that diamond properties assertions are created, they must specify the pre and post-conditions and define equivalence conditions. In addition, the assertion must execute the operations in two different orders, verify if the constraints are not violated, and check if both executions are equivalent.

2.2.4 Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture [2]

In addition to the introduced Output-Based commutativity (Section 2.1.5) and Random Interpretation (Section 2.1.6.5), it is presented a system that aims to parallelize a program by executing concurrently individual functions that are commutative. It is presented in Object-Oriented Programming, which involves working on objects. Commutativity analysis is performed during compile time using the introduced Random Interpretation, and it is built under the Output-Based type.

It proposes an algorithm that automatically identifies commutative functions. It does not test if memory contents are precisely equal. Instead, it only checks if the outputs of a program are correct if reordering the operations in some arbitrary order.

The idea is that as long as the different results created by changing the execution order do not affect the program output, then the compiler is free to reorder the functions. The two main technical challenges of this process are checking the equivalence of the executions and identifying the set of functions that read another function's output.

Identifying the readers of a function's results is simply a matter of data-flow analysis. However, identifying readers of functions that produce side effects is harder. Points-to analysis determines the set of all possible addresses a pointer could refer to. Some pointer analyses are relatively conservative. They can differentiate global variables and stack accesses but cannot differentiate heap objects. So Fulcra [31], which has support for differentiating heap objects, can be used for more precise results.

To check the equivalence of the executions, previously introduced Random Interpretation is used (section 2.1.6.5), so two different parallel executions using Random Interpretations starting from the same initial state are issued, one with the regular execution order, and the other with the functions that are being tested test reordered. If the two runs produce the same output, then the tested functions commute.

A limitation is that the algorithm requires the full vision of the program. Note that two operations can be commutative if executed sequentially before an output operation. However, they are not commutative if the output operation executes in the middle of the two functions.

2.2.5 COMMSET [33]

In this work, the programmer must specify which blocks of code commute and under which conditions, which means that the programmer decides which operations will be analyzed. At compilation time, the compiler will verify if the conditions for each block are met, and it assumes that code blocks can commute only by meeting the conditions. This requires an extra effort for the programmer to reason about which blocks of code should be analyzed and which conditions should be defined. A wrong specification may adulterate the program's behavior. It is described in Imperative Programming and permits working on non-primitive data types such as pointers or arrays.

Commutative Set (COMMSET) is an implicit parallel programming model based on semantic commutativity. In Implicit parallel programming models, programmers implicitly expose inherent parallelism in their program without explicitly using low-level parallelization constructs. Instead, programmers express existent parallelism via the use of extensions to the sequential model. The COMMSET extensions are expressed using pragma directives in the sequential program.

A COMMSET is a set composed of blocks of code where blocks that belong to the same COMMSET commute. A program may have multiple COMMSETs, and a code block can belong to many COMMSETs. There are two possible COMMSET types, Self, where a block is self-commutative, and Group, where blocks that belong to the same COMMSET commute between them. In addition, a predicate can be associated with a COMMSET. When two blocks that belong to a COMMSET are being tested, the predicates associated with that COMMSET will be checked. Then two members commute if all the predicates evaluate to be true.

Initially, the system parses and checks the syntax of all COMMSET directives and

synthesizes a function for every COMMSET predicate. Blocks of code associated with a COMMSET are converted into functions, and predicates are checked to verify if they are deterministic. A Program Dependence Graph(PDG) is built with its respective edges dependencies, which are classified as unconditionally commutative or inter-iteration commutative. In the next step the system will execute DOALL [3] and PS-DSWP [34] , which automatically partitions the PDG onto multiple threads. Finally, synchronization primitives are inserted automatically to ensure atomicity.

2.2.6 Hamsaz [21]

Hamsaz is a system that analyses operations on a distributed system. It analyses operations commutativity by running an SMT solver during compile time. The programmer must describe the effects and invariants of the operations. Then the solver uses those definitions to check commutativity. The presented examples are simple methods that alter class variables. In the end, only the final state of each variable is analyzed. So we classify the Hamsaz analysis type as separability-based.

Hamsaz's objective is to synthesize a replicated object automatically. The state of the object is replicated across replicas. Clients can call methods at every replica, and the calls are communicated between replicas. The replicated object is expected to satisfy both consistency and convergence. Consistency is the safety property that every method call is executed only when the guard of the method and the invariant are satisfied. Convergence is the safety property in which replicas converge to the same state when no call is in transit.

One of the key points is to perform coordination only when necessary to preserve these properties. The system uses commutativity to avoid unnecessary coordination and to detect conflicts. The input to Hamsaz is the object definition, which includes the state type, invariants on the state, and methods.

The first phase of Hamsaz determines the conflicts and dependencies between operations.

Two method calls S-commute (state-commute) if starting from the same pre-state, executing them in either of the two orders results in the same post-state. Otherwise, we say that they are S-conflict (state-conflict) and need synchronization, which implies that method calls should be executed one at a time.

Methods that never violate invariants are called invariant-sufficient methods.

P-R-commutes (permissible-right-commutes) require that the method call c_1 stays permissible if moved to the right/after c_2. A method call c_1 P-concurs (permissible-concurs) with another call c_2 if either c_1 is invariant-sufficient or c_1 P-R-commutes with c_2. Otherwise, we say they are P-conflict (permissible-conflict) and need synchronization. Two method calls concur if they both S-commute and P-concur with each other.

A method call c_2 P-L-commutes (permissible-left-commutes) with a method call c_1 if c_2 remains permissible if it is moved left/before c_1. We say that a method call c_2 is

independent of c_1 if c_2 is either invariant-sufficient or P-L-commutes with c_1.

A static analysis is used to determine the conflicts and dependencies with the help of an SMT solver to decide the validity of concur and independence relations for every pair of methods. First, it calculates the S-commutativity relation. Then calculate the invariant-sufficiency relation and P-R-commutes, which will be used to calculate the P-concur relation. After, the concur relation is calculated from S-commutativity and P-concur. The conflict relation is calculated as the negation of the concur relation. Finally, P-L-commutativity is calculated, which is used in addition to invariant-sufficient to calculate the independence relation and obtain the dependence relation from the negation of the Independence relation.

A synchronization protocol is necessary to ensure consistency and convergence in case of conflicts or dependencies. Analysis results are used to instantiate the protocols, and the second phase starts. Two types of protocols are presented, a Non-Blocking protocol and a Blocking protocol. The main idea of the Non-blocking Protocol is to find sets of conflicting calls and synchronize calls in each set. The classical total-order broadcast (TOB) protocol is used to deliver method calls in the same order at all replicas. The conflict call is broadcast to each TOB instance and executed only when it is ordered and delivered by all TOB instances.

The main idea of the Blocking protocol is to avoid synchronization in both methods but only synchronize one of them. Two conflicting methods, c_1 and c_2, and c_1, executes without synchronization. The idea is that calls on c_2 reach out to other replicas and block the execution of calls on c_1, but replicas exchange updates on preceding calls on c_1. After all, replicas apply the updates and have the same set of executed calls on c_1. c_2 is executed at all replicas, and calls on c_1 are unblocked.

2.2.7 Servois [6]

Servois is an auxiliary automated tool that synthesizes commutativity conditions and consequently evaluates if operations are commutative. It is introduced for evaluating commutativity conditions for data structures, "objects", and Smart-Contracts.

The user should specify operations, effects, pre and post-conditions in an auxiliary YAML file. Additionally, the user should specify terms to help the program generate condition predicates. Usually, pre and post-conditions express conditions where operations do or do not conflict. These condition predicates will be used to test commutativity.

The core phase is the refine phase, where operations are tested recursively. Refine phase tests a predicate by trying to find a counter-example where that predicate is not enough to ensure commutativity. It will test for commutativity and non-commutativity. So a predicate will be tested for commutativity, and if no counter-example is found, that predicate guarantees commutativity. At the same time, it will also be tested for non-commutativity. At least one of them(commutativity or non-commutativity) must

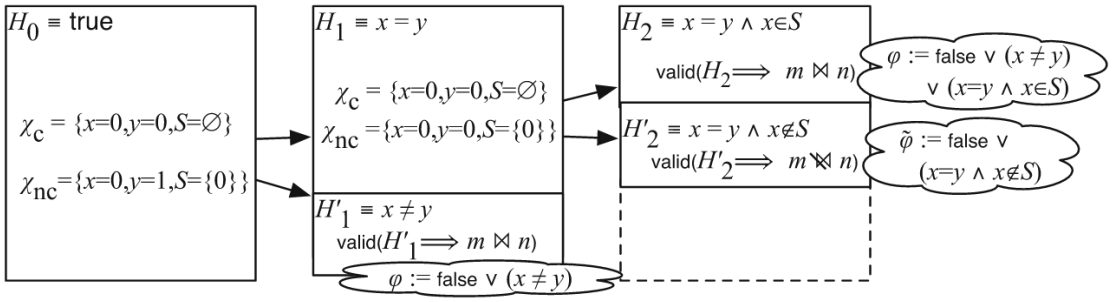

Figure 2.3: Servois commutativity analysis, Add and Contains in a Set(from[6])

have a counter-example. If there is a counterexample for commutativity and for non-commutativity it means that the evaluated predicate is not suffice to determine commutativity. Thus it is necessary to add another predicate to the condition and reevaluate the new condition(conjunction of the previous predicate with a new one). If there is no counter-example for non-commutativity, then the operation is non-commutative. Otherwise, if there is no counter-example for commutativity, then the operation is commutative.

At the start, there is no predicate, only a true value, and if there is no counter-example for commutativity, then operations are always commutative, if there is no counter-example for non-commutativity, then operations are always non-commutative. If there are counter-examples for commutativity and non-commutativity, a new predicate is chosen, and it will be tested. This process repeats while a decision is not taken.

There is a small challenge in choosing the new predicate. The number of generated predicates grows exponentially with the number of terms the user suggested, so trivial predicates are discarded. Next, it only chooses predicates that fit in the previous counter-example and are distinguished [17](predicates that mean the opposite ex: $x = y$,$x \neq y$). Now both predicates are tested, as explained before. If conditions are not precise, the number of tested predicates grows exponentially. So a technique that looks one step ahead while choosing the predicate is called poke, and it will execute the next iteration using the candidates after it counts how many distinguished predicates remain. Finally, it will choose the predicate that "originates"less distinguished predicates.

An example is Add and Contains operations from a Set (figure 2.3). In the beginning, no predicates are applied, only a true condition. X_c is a counter-example for commutativity and X_{nc} a counter-example for non-commutativity. In the first box, both have counter-examples, so two distinguished predicates are chosen. Now both predicates will be tested in the second box. For $x = y$, we have two counter-examples again, so two new distinguished predicates are chosen, and when $x \neq y$ no counter-example for commutativity is found so $x \neq y$ is a condition where operations are commutative. We should proceed to the third box, where another distinguished predicate is tested. In this case, $x = y \wedge x \in S$ is commutative (has no counter-examples for commutativity), and $x = y \wedge x \notin S$ is non-commutativity(has no counter-examples for non-commutativity).

As we want to achieve commutativity, Add and Contains are commutative if ($x \neq$

$y) \vee (x = y \wedge x \in S)$

2.2.8 Loop Parallelization using Dynamic Commutativity Analysis [37]

DCA combines both static and dynamic information aiming to parallelize sequential code. It focuses on loop parallelization, as loops are regions of code that usually take a considerable execution time. It tests commutativity by executing the regular version and comparing it to a set of random executions using random values. It is classified based on liveness since only the final output is required for commutativity verification. This work is presented in an Imperative language and may work on non-primitive data types such as structures or pointers.

First, it initiates with a static phase, starts by looking for variables that will be updated on each iteration, and determine if execution continues in the loop body or exits out of it. In the second phase, it will take the loop payload(functions inside the loop), export the payload to a separate function, and identify live-in and live-out variables. After Iterator Linearization is executed, it has identified the iterator code to extract its values. This process linearizes subsequent accesses to the values of the iterator. The final static stage is Commutativity Testing Instrumentation, where commutative permutations and properties verification are defined.

The dynamic stage starts by recording the normal execution of the loop, then a set of permutations is selected, which will control the exact reordering of the loop iterations. A standard execution order loop is executed, and the other permutations are exhaustively executed. A random subset is used because it is impossible to test all possible permutations. This means accepting a chance of missing a permutation that does not commute. In the end, live-out values will be compared. If the output of at least one permuted execution differs from the original output, then the loop is not commutative.

2.2.9 Discussion

By analyzing the presented solutions when looking at how the analysis is performed, it is observed that a large part of them recur to SMT/SAT solvers, and others rely on symbolic execution. In the case of CASP [2] and DCA [37], which use a random analysis, although they have a very high probability of correctness, some corner cases may not be analyzed because random values are used, and the analysis may return wrong results.

It is possible to observe that most solutions use a compile-time analysis, and only one is analyzed during runtime. Once the context of our problem is the distributed systems, the main objective is to reduce the latency. If the analysis was performed during a method call (runtime), it would increase the latency. Thus, relying on a compile-time analysis is vital to mitigate the latency increase.

When analyzing the type of commutativity, we discard the separated-based because we would be restricted to an analysis that required the same methods to be invoked. Even if the methods had the exact instructions, but they were different methods, they would

	Analysis	Type	Data Types	Context/Scenario	Paradigm	Developer Input
SFCA [35]	Compile/Symbolic	Separated	Objects	Separable Methods	OOP	None
ESCAC [39]	Compile/SMT	Separated	Objects	Methods calls	OOP	None
ACADL [15]	Aux Tool/SAT	(Output)	Objects	Queue	OCL	Effects and Conditions
CASP [2]	Compile/Random Interpretation	Output	Objects	Individual Functions	OOP	None
COMMSET [33]	(user defined)	(user defined)	(Non-)Primitive	Blocks of Code	Imperative	Conditions and Code Blocks
Hamsaz [21]	Compile/SMT	Separated	Objects	Method calls	OOP	Effects and Conditions
Servois [6]	Aux Tool/SMT	(Output)	Strucutres	Data Structures	OOP	Effects and Conditions
DCA [37]	Runtime/Random Testing	Liveness	(Non-)Primitive	Loops	Imperative	None
Our proposal	Compile/Symbolic	Liveness	Objects/Primitive	Method calls	OOP	Conditions

Table 2.2: Commutativity Parallelization

not commute. The Output based would solve this problem, but for operations that do not require consistency, it would fail once it produces different outputs. Thus, we adopted the Liveness type, which permits more output freedom but consistently maintains the values of the replicated fields.

Analyzing some of the existing solutions makes it possible to conclude that five of the solutions tend to analyze commutativity on objects. However, when they talk about objects, they refer to an abstraction or a struct (a class that contains fields but does not support polymorphism or inheritance). Thus, when referring to objects, they do not consider the object references and sub-objects.

Additionally, the solutions that rely on SMT/SAT solvers require that the developer defines the effects (the expressions that the solver will evaluate). Sometimes, these definitions are complex, and an error from the developer may result in incorrect results. Thus, the definition of expressions to be evaluated will be extracted automatically from the source code.

After taking all the knowledge from the different concepts, we propose a solution that relies on a compile-time analysis, wherewith symbolic execution, where the effects expressions are extracted. Then they will be evaluated using the SMT solver. It will rely on a liveness base solution where the response to the request can differ, but the system's state will be consistent. Furthermore, it will support simple objects with subfields, references, and primitive types. This solution will be applied to Java, which is included in the OOP paradigm.

3

Solution Overview

In this chapter, we will briefly describe the developed solution used to evaluate if two method calls commute and create lambda expressions with the conditions where these two method calls commute. We will start by describing the different stages from the built model and succinctly describing each stage. Finally, we will present some considerations the developer must have to achieve a correct execution.

3.1 Solution Description

The model can be divided into three stages. Consider the Figure 3.1, which represents the solution overview. It is possible to observe that the first stage is the *side effects extraction*. The *side effects extraction* consumes the program's bytecode, and visits all the class methods. For each method, it extracts the side effects, the field invariants, and method pre-conditions. The *commutativity analysis* stage consists in analyzing the side effects, invariants, and pre-conditions extracted from the previous stage. Then, it evaluates if two method calls commute and defines which are the necessary conditions for the method calls to commute. Lastly, the final stage creates the *commutativity verification map* or *condition map* that will be the output of our solution. This map will be made available to the runtime system, so that the latter may rapidly verify if two method calls commute or not. To create the map it is necessary to parse the conditions returned by the *commutativity analysis* and parse them to an executable function.

Figure 3.1 is a general overview, but in each of the three stages there are smaller inner stages. Thus we will describe the three stages in more detail, but the implementation details will only be presented in the next chapter.

3.1.1 Conditions/Invariants Syntax

Before we dive into the details of *side effects extraction* stage, it is necessary to show some characteristics presented in the solution. As shown in the Account Java example of Listing 1 in Chapter 1, we defined some concepts associated with the field or the method. Two of these concepts are the invariants associated with the field and represented by the

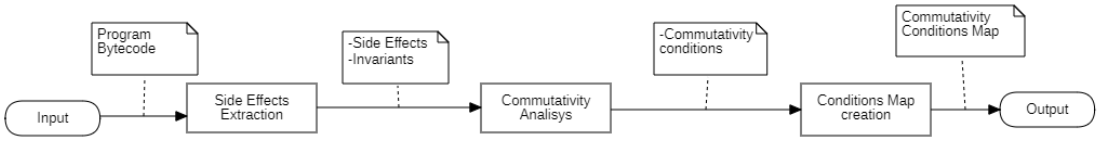

Figure 3.1: Commutativity analysis and condition extraction pipeline.

Expr	::=	Expr Op Expr | ! Expr | (Expr) | AExpr
Op	::=	> | < | == |!= | <= | >= |&& ||
AExpr	::=	AExpr AOp AExpr | Identifier | Literal | MCall
AOp	::=	+ | − | ∗ | /
MCall	::=	Identifier . Identifier(Expr)

Table 3.1: Grammar definition for condition expressions

`@Invariant` annotation and the method pre-conditions associated with the method and represented by the `@MethodCondition` annotation. Both of them require an expression to define the condition. Thus, defining these expressions correctly is crucial because the wrong syntax may break the program execution. The syntax for the expression conditions is based on the Java syntax and follows the grammar defined in Table 3.1 .

It is also vital to notice that when passing conditions involving parameters, the parameter definition on the condition should not be the parameter descriptor, but instead the word ***param*** followed by the parameter's index. From the bytecode analysis, the information we extract for method parameters is only the indices and their types, so once we do not have the descriptor information, we have decided to discard the descriptor for parameter identification.

So, for example, to define a method condition that has the objective to impose that the amount value of a withdrawal must be lower than the balance, it should be expressed as follows:

```
@MethodCondition(expression = "param0 < balance")
public void withdraw(int amount) {
    this.balance -= amount;
}
```

3.1.2 Effect Extraction on Consistent Reads

Some methods do not produce side effects, for instance, a *gettter* method that only returns a field's value and does not change it. If a method does not produce side effects, it commutes with all other methods because only these other methods may affect the variables. However, a replicated system may be obligatory to ensure reading consistency. As a read operation does not produce effects, it always commutes and thus can return

a non-consistent value. An annotation at the method level (@ConsistentRead) was created to define that the annotated method required consistency. This way, it is possible to extract a fictitious side effect, which during commutativity analysis is evaluated.

What would be the side effect expression to express the consistency effect on a read method? If we say that the new field value is equal to the old one, it would result in an idempotent effect. Independent of the old value, it would maintain that value. Thus, the method would commute with all methods. Imagine the evaluation of method call *deposit(10)* and method call *getBalance()* with an initial *balance* of 30 where both methods access the field *balance*. If the method call's order is first *getBalance()* and then *deposit(10)*, the final *balance* is 40. If the method call's order is inverted, the final result will also be 40. Thus,remembering the commutativity property of equivalence defined in Section 2.1.2 the method calls commute. However, the user wants consistency in the read operation over *balance*. In that case, these operations should not commute because if the *getBalance()* method call is executed first, it will return 30, and if it is executed after, it will return 40.

The adopted solution was to define an auxiliary variable representing the original field. This auxiliary variable is only updated on the consistent reads for its original field. Thus, when a consistent read is required, the auxiliary variable needs to update its value to the current value of the original field. So, the auxiliary variable is forced to have the most recent value, and when evaluating commutativity, the equivalence state is not reachable if the original value has changed. The variable comprises the prefix *read_* and the original field's name.

So, with the creation of the auxiliary variable, resulting in *read_balance* when evaluating commutativity, if the method call's order is first the read and then the deposit, it results in *read_balance* = 30 and *balance* = 40, on the other hand, if the method call's order is inverted, the final result is *read_balance* = 40 and *balance* = 40, which violates the equivalence property, because the final state is different. Thus, the two methods do not commute. This way, it ensures that the read operations are executed in the proper order and that the returned value will be consistent.

Taking the class Account presented in Section 1, to make the `getBalance()` method with strong consistency requirements, it is only necessary to add the `@ConsistenRead` annotation as shown in Listing 2.

An effect is defined by the left side, which is the field that is being affected, and the right side, which is the expression that is changing the field. The side effect produced for the consistent read is *read_balance* → *balance*. It means that the *balance* value is being assigned to the *read_balance* field. In this example, the auxiliary field is created for the consistent reads.

3.1.3 Side Effects Extraction

The first step is to extract side effects from each method. So, the visitor starts by visiting the class methods. As mentioned before, fields can have the property of being replicated

```
@ConsistentRead
float getBalance () {
    return this.balance;
}
```

Listing 2: Consistent read annotation for the `getBalance()` method of the Account class in Listing 1

among all the replicas by the use of the `@Repl` annotation. Our solution only extracts side effects for replicated fields. Thus, it is essential to identify correctly the fields that should be replicated to extract and evaluate commutativity correctly.

The bytecode visitor visits all the instructions inside the method. It analyses each instruction and its operands. Depending on the instruction type, we extract effects or not. If the instruction is an invocation to another method, it is created a `CallEffect`. If the instruction is a field assignation to a replicated, an `Effect` is created. The extracted side effect created in `Effect` applied to a field is the right side expression of the equality $field = expression$. If the instruction is a replicated field return and is from a method that has the property consistent read (introduced in the previous section) it is created an `Effect`. As reads do not produce side effects, the created `Effect` introduces an auxiliary field which is going to be equaled to the returned field. The created `CallEffects` and `Effects` are stored in the method's `EffectsList`. The stored effects of type `CallEffect` will be resolved later in an inter-procedural analysis.

The `EffectsList` is a list that stores all the method effects. Thus, each method has its own `EffectsList`. The `EffectsList` is composed of the `EffectInterface` interface objects(Listing 3).

The Effect class is composed of two Resources, the left and the right resources. A Resources is an object representation in memory. There are different types of resources, such as `ConstResource`,the `FieldResource`, the `ObjectResource`, the `ParameterResource`, the `BinaryOperationResult` and the `StoreOpResult` which are explained with detail in the next chapter (Chapter 4.1.1). The left resource represents the field that is being affected. The right resource represents the effect expression. The CallEffect class represents a method call, which will be analyzed later and may be converted into an Effect. The CallEffect is composed of the calling class and method, the target class and method, and the parameters used on the method call, which are Resources.

Taking as an example the Account class (Listing 4), the produced effects are the ones depicted in Table 3.2. For the `deposit` method, the generated is an effect once it is a simple field assignation where the *balance* field is equal to itself plus the amount (the parameter with index zero). The process is the same for the `withdraw` method but subtracts the amount from the *balance* field instead of summing it. The `transfer` is more interesting once it does not have field assignation but has method calls. Thus, as mentioned

```
public interface EffectInterface {}

public class CallEffect implements EffectInterface {

    private String callingClassName;
    private String callingMethodName;
    private Resource targetClass;
    private String targetMethodName;
    private List<ResourceReference> parameters;
    ... // other methods
}

public class Effect implements EffectInterface {

    protected Resource left;
    protected Resource right;
    ... // other methods
}
```

Listing 3: Effect and CallEffect classes

deposit	$balance \rightarrow balance + amount$
withdraw	$balance \rightarrow balance - amount$
transfer	$callEffect(this, withdraw)$ $callEffect(to, deposit)$
accrueInterest	$balance \rightarrow balance + (balance * amount)$
getBalance	–
getConsistent_Balance	$read\_balance \rightarrow balance$

Table 3.2: Effects after primitive extraction on Account class

above, it produces a `CallEffect`, which will be resolved later. The `getBalance` method does not produce effects once it is a read, and reads do not produce effects. Finally, the `getConsistent_Balance`, which is also a read but has the `@ConsistentRead` annotation, indicates that the method must return a consistent value. In order to ensure consistency in commutativity analysis, the method needs to produce side effects. Thus, the auxiliary field,*read_balance* , is used for consistent reads.

One may notice that in Table 3.2, the produced effects by the method `transfer` are not so simple because they contain method calls that have not been resolved yet. Thus, it is necessary to analyze the method calls and extract their effects.

To resolve the method calls, for each method, we visit its `EffectsList`, and if it is present a `CallEffect` in the list, we resolve it. From the `CallEffect`, we acquire the called method and called class. In a first phase, we visit the called method's `EffectsList` and add all its `Effects` to the calling method's `EffectsList`. Additionally, it is necessary to do some mappings for the `Effects` that come from the called method. These `Effects`

```java
public class Account {
    @Repl
    @Invariant("balance >= 0")
    private int balance;

    @MethodCondition(expression = "param0 > 0")
    public void deposit(int amount) {
        if (amount > 0)
        this.balance += amount;
    }

    @MethodCondition(expression = "param0 > 0")
    public void withdraw(int amount) {
        this.balance -= amount;
    }

    @MethodCondition(expression = "param1 > 0")
    public void transfer(Account to, int amount) {
            withdraw(amount);
            to.deposit(amount);
    }

    @MethodCondition(expression = "param0 > 0")
    public void accrueInterest(int interestRate) {
        if (interestRate > 0)
        this.balance = this.balance + this.balance * interestRate;
    }

    public int getBalance() {
        return this.balance;
    }

    @ConsistentRead
    public int getConsistent_Balance() {
        return this.balance;
    }

}
```

Listing 4: Account class REPL-Java (same as in Section 1)

can have parameters, and mapping them to the values passed by the calling method is necessary. If the method call target is an object, we verify if the object is replicated or has replicated fields. If the target has replicated fields, we apply the same process mentioned before but with a slight change. After the creation of the `Effect`, a parent relation is assigned using the target object, informing that the `Effect` is affecting the object and then is added to the calling method's `EffectsList`. When the object is replicated, any

deposit	$balance \rightarrow balance + amount$
withdraw	$balance \rightarrow balance - amount$
transfer	$balance \rightarrow balance - amount$ $to.balance \rightarrow to.balance + amount$
accrueInterest	$balance \rightarrow balance + (balance * amount)$
getBalance	–
getConsistent_Balance	$read\_balance \rightarrow balance$

Table 3.3: Effects after method calls resolution on Account class

change in the object is an effect. Thus, we need to understand if the called method has any effect even if it is not on a replicated field. So, during effect extraction from bytecode, all the possible side effects for all the fields are stored even if they are not replicated. Now, the object retrieves the information and, in case of a side effect for the called method, creates an `Effect` with the target object and is added to the calling method's `EffectsList`. In the end, the `CallEffect` that has been processed is removed from the `EffectsList`.

After resolving all the dependencies, the previously presented effects on Table 3.2 become those in Table 3.3. The `transfer` method that in the Table 3.2 only had `CallEffects`, in Table 3.3 it already has their corresponding effects. The $callEffect(this, withdraw)$ effect resorts to the most straightforward type of dependency, and it was only necessary to obtain the `EffectsList` from the `withdraw` method, extract the effect's and map them according to the `CallEffect` parameters. Thus, from the `withdraw` method, we obtain $balance \rightarrow balance - param0$, which maps the $amount$ parameter to the $param0$, resulting in $balance \rightarrow balance - amount$. The $callEffect(to, deposit)$ is a little bit harder because it is applied to an object. Thus, it is necessary to extract the `EffectsList` from the deposit, which has the $balance \rightarrow balance + param0$ effect. So, it is necessary to map the parameters and attribute the parent relation. The effect is $to.balance \rightarrow to.balance + amount$.

3.1.4 Commutativity Analysis

In our solution's second stage, we evaluate commutativity using the previously extracted side effects. We resort to Servois as our solution solver once it is capable of evaluating if the method calls commute and it is capable of defining the commutativity conditions. Servois has in its core, an SMT-Solver, which takes as input SMT-Lib expressions. Thus, the next step is to define expressions in the SMT-Lib syntax.

The class methods information and the effects are parsed to SMT-Lib expressions. As defined in the Section 3.1.1, the expressions from the `@Invariant` and `@MethodCondition` annotations are extracted. These expressions, which have a similar Java syntax, need to be parsed to SMT-Lib expressions. JavaCC [24] which is a parser generator that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar, helps us to parse the expression to SMT-Lib expressions.

After the definition of mappings between our solution and SMT-Lib, the SMT-Lib expressions for the extracted effects are defined. Table 3.4 presents the effects from

deposit	(= *new_balance* (+ *balance amount*)) (= *new_others.balance* (*others.balance*)) (= *new_read_balance read_balance*)
withdraw	(= *new_balance* (− *balance amount*)) (= *new_others.balance* (*others.balance*)) (= *new_read_balance read_balance*)
transfer	(= *new_balance* (− *balance amount*)) (= *new_others.balance* (+ *others.balance amount*)) (= *new_read_balance read_balance*)
accrueInterest	(= *new_balance* (+ *balance* (∗ *balance amount*))) (= *new_others.balance* (*others.balance*)) (= *new_read_balance read_balance*)
balance	(= *new_balance* (*balance*)) (= *new_others.balance* (*others.balance*)) (= *new_read_balance read_balance*)
getConsistent_Balance	(= *new_balance* (*balance*)) (= *new_others.balance* (*others.balance*)) (= *new_read_balance balance*)

Table 3.4: Effects on SMT-Lib expressions

Table 3.3 in the SMT-Lib format. All methods include the value to be assigned to all of the class' fields, even if the latter are unaffected by the method. This is a Servois restriction that requires all the existent fields to be defined in all the methods. So, for fields that do not have a side effect in a method, it is necessary to define its effect. So, for each field we create a new effect that is idempotent by saying that its new value is equal to the previous. This allow us to define an effect o the field without changing its value. For example, this behavior can be observed in the `getBalance()`, where the field *balance* does not produce any effect, but we have to define its idempotent effect, which is defined as (= *new_balance balance*). The objects that are not from the class, i.e., they are object parameters, are viewed as external entities. Thus, in the parsing, we define them with the *others* prefix.

After the parsing of the side effects to SMT-Lib, the Servois YAML file is created and it is written on disk. Then when executing Servois, the file is consumed. The entire YAML file for the Account example is presented in the Annex II.1. At the end of Servois execution, a results file is created and written on disk. The resulting file (Listing 5) consists of multiple lines where the method pair is identified as the two first strings and the commutativity condition after(third string). The commutativity condition can be ***true*** if the method pair always commute, can be ***false*** if the method pair never commute, can be ***false_un*** which means that methods do not commute but this is assumed because Servois could not evaluate commutativity, and finally can return an SMT-Lib expression.

Our solution reads the Servois results and for each line it needs to parse the SMT-Lib expression to a syntax to be converted to a java Function. Thus, this parse is again done with JavaCC help, which parses the Servois conditions to a string with JavaScript syntax.

```
accrueInterest accrueInterest true
accrueInterest deposit false
accrueInterest getBalance true
accrueInterest getConsistent_Balance (not (> balance 0))
accrueInterest transfer false
accrueInterest withdraw false
deposit deposit true
deposit getBalance true
deposit getConsistent_Balance false
transfer deposit true
deposit transfer (not (> y2 balance))
withdraw deposit true
deposit withdraw (not (> y1 balance))
getBalance getBalance true
getBalance getConsistent_Balance true
getBalance transfer true
getBalance withdraw true
getConsistent_Balance getConsistent_Balance true
getConsistent_Balance transfer false
getConsistent_Balance withdraw false
transfer transfer (or (and (> y2 (- balance x2)) (> (+ OTHER.balance y2) (- balance x2))
                  (not (> x2 balance)) (not (> y2 balance))) (and (> y2 (- balance x2))
                  (not (> (+ OTHER.balance y2) (- balance x2))) (not (> x2 balance))
                  (not (> y2 balance))) (not (> y2 (- balance x2))))
withdraw transfer (not (> y2 balance))
transfer withdraw (not (> y1 balance))
withdraw withdraw (or (= (- balance x1) y1) (and (not (= (- balance x1) y1))
                  (not (> x1 balance)) (not (> y1 balance))))
```

Listing 5: Servois Account Results

The JavaScript syntax is adopted because we use the nashorn [30] engine to create the Java functions and the nashorn engine takes Javascript syntax Strings as input.

3.1.5 Commutativity Verification Map Creation

The last stage of our solution has the purpose of creating the commutativity verification map. It receives a result of the solver as a sequence of strings in JavaScript syntax. These are parsed into Java Bifunctions [22], which take two arrays of objects (`Object[]`) as arguments and produce a Boolean: `BiFunction<Object[], Object[], Boolean>`, i.e. functions of the type `(Object[], Object[])` $\rightarrow$ `Boolean`. The resulting functions are added to map.

To parse the JavaScript syntax into Java Functions we make of the Nashorn [30] JavaScript engine.

```
public interface MethodEffect {
    EffectsList getEffect(CallEffect callEffect, MethodCall methodCall);
}
```

Listing 6: MethodEffect Interface Implementation

```
public interface ManualStructureInterface {
    public String className();
    public String servois_Constructor();
    public MethodEffect methodEffects(String methodSignture);
}
```

Listing 7: ManualStructureInterface Implementation

3.2 Specification of Effects for Frequent Types

There are some classes that are very recurrent, or the effects are tough to extract. To optimize the performance of the side extraction or define side effects that are hard to obtain, two interfaces (`MethodEffect` and `ManualStructureInterface`) were created that permits the programmer to define the side effects for each method of that class.

The `MethodEffect` interface (Listing 6) only has a method, `getEffect`, which receives a CallEffect and the correspondent MethodCall and returns an `EffectsList`.

The `ManualStructureInterface` interface (Listing 7) comprises three methods. The `className()` must return the name of the class that we are defining the `Effects`. The `servois_Constructor()`, which must return the Servois constructor declaration. Servois constructor declaration is defined by the class type and its field types both in SMT-Lib surrounded by parenthesis, as follows:

$$(< class\_type > < field\_type_1 > < field\_type_2 > \cdots < field\_type_n >)$$

The last method is `methodEffects(String methodSignature)`, which, given a method signature, returns an interface of type `MethodEffect`.

Our solution already supports some methods of the Set and Map classes. For instance, Listing 8 shows an implementation of the `MethodEffect` for the `Add` class, which represents the add method from the Set. The class only has the method required by the MethodEffect interface and an empty constructor. In line 5, the `EffectsList` object, which is returned at the end of the method, is created. As defined in Subsection 3.1.3, the `Effects` are composed by Resources.

To define an `Effect` it is necessary to define the field that is being affected. In line 6, from the methodCall object, we obtain the target of the call, in this case the Set where we are adding the new object. In line 7, we create a Boolean `ConstResource` with the

```
1  public static final class Add implements MethodEffect {
2
3    @Override
4    public EffectsList getEffect(CallEffect callEffect, MethodCall methodCall) {
5        EffectsList ef = new EffectsList();
6        Resource set = methodCall.getTarget().resource;
7        Resource bool_true = new ConstResource<Boolean>(PrimitiveType.Boolean,
         ↪  Visibility.LOCAL, true);
8        StoreOpResult storeResult = new StoreOpResult(set,
         ↪  callEffect.getParameters().get(0).resource, bool_true);
9        ef.add(new Effect(set, storeResult));
10       return ef;
11   }
12 }
```

Listing 8: MethodEffect Class example implementation

true value. Once SMT-Lib does not directly support the use of sets, we need to go around. Thus, a set in the SMT-Lib is a map where the key is the object we pretend to alter in the set, and the value is a Boolean. If it is false, it means the object does not exist; if it is true, the object exists. Thus, in line 8, we create the `StoreOpResult`, which is applied on the method call target, the *set*. Once the `CallEffect` has the parameters from the method call, the object we desire to add to the set comes from the `CallEffect` parameters. Thus, the object we are adding to the set is the parameter extracted from `callEffect.getParameters().get(0).resource` expression. Finally, once we add the object to the set, we define the value as true. In line 9, we create the `Effect`, by defining the left resource to be the `set` and the right resource to be the `StoreOpResult`, we add it to the `ef` effect's list.

After showcasing the implementation of a `MethodEffect` object, we show an implementation example of the `ManualStructureInterface` interface in Listing 9.

The class features a map, `methodEffects`, with the effects defined for each method of the set (line 2). The class' constructor fills this map with the effects of all methods for which effects have been specified. Line 5 shows the proceeding for method with signature `add(java.lang.Object)` by using the class presented in Listing 8.

In line 9, method `className` returns the class' name. In this example, it is applied to the `java.util.Set` so it returns the *java.util.Set* string.

In line 10, method `servois_Constructor` returns a string with the Servois constructor for the class. It respects the SMT-Lib syntax where the type is identified, in this case, the *Set*. Once a Set can be of different types, we define *%s* to represent an abstract type. Our solution automatically identifies the used type and substitutes the *%s* for the corresponding SMT-Lib type. If, for instance, we want to define a set of integers, the Servois constructor is (*Set Int*). Another detail for the Servois constructor is that the

```java
public class JavaSet implements ManualStructureInterface {
    private Map<String, MethodEffect> methodEffects = new HashMap<>();

    public JavaSet() {
        methodEffects.put("add(java.lang.Object)", new Add());
        ... // other methods
    }

    public String className(){return "java.util.Set";}
    public String servois_Constructor(){return "(Set %s)";}
    public MethodEffect methodEffects(String methodSignture) {
        return methodEffects.get(methodSignture);
    }
}
```

Listing 9: Set Manual Specification Interface implementation

```java
public interface RepositoryInterface {
    ManualStructureInterface getManualStructure(String className);
}
```

Listing 10: Manually Defined Types Repository Interface

parenthesis must surround it.

The last method (lines 11 to 13) returns the `MethodEffect`. From the method signatures, it retrieves the `MethodEffect` from the *methodEffects* Map.

With these simple steps, explained in the Set example, the side effects can be defined for every class the developer pretends.

In the end, the solution must know which classes are implemented using the `ManualStructureInterface` or be able to extract the manually defined side effects. Thus, we have defined an interface that works as a repository for classes that implemented the `ManualStructureInterface`. So our solution can consult the repository and obtain the implementations of the `ManualStructureInterface` for the desired class. The `RepositoryInterface` is a simple interface that requires the implementation of a method that receives the class name and returns `ManualStructureInterface` object for the corresponding class name (Listing 10).

3.3 Mapping from Java Types into SMT-Lib Types

As our current solution resorts to Servois, which receives as input SMT-Lib expressions. The SMT-Lib is not as extensive as Java and has its own way of defining types, which

```
int = Int
boolean = Bool
float = Real
java.lang.integer = Int
integer = Int
double = Real
java.lang.double = Real
long = Real
java.lang.Long = Real
java.lang.string  = String
string  = String
java.util.uuid = String
uuid = String
```

Listing 11: Java to SMT-Lib types mapping

differs from Java. Hence, in order to have our internal representations translated into SMT-Lib's syntax it was necessary to develop a translator.

In order to not be limited to the existing translations and extend the possible used types for our solution, we define a properties file to define the type's conversion.

The properties file has, as key, the type in Java and, as value, the correspondent value in SMT-Lib. To support the most used basic types, the properties file is already filled with the conversion for ints, strings, doubles, floats, longs, and booleans. It is also possible to convert more complex types, such as Java UUID, and present them in more straightforward types, such as a string in the following properties file.

In case of adding more types, it is only necessary to follow the logic shown in Listing 11. The Java type is on the left side, and the SMT-Lib type is on the right. One small detail is that it is necessary to define the Java type with the entire object type (ex: Java.lang.string) and the simple form (ex: string).

3.4 Building the Commutativity Verification Map from Servois' Output

Servois does not return a condition for all the method pairs in both directions. If the condition is equal in both executing orders Servois only returns the condition for one of the executing orders. If if `method1` commutes with `method2` in some condition C and `method2` commutes with `method1` in the same condition C, the Servois output is only that `method1` commutes with `method2` in condition C. And does not return any information about the conditions for the `method2` `method1` execution order.

Thus when building the map with the conditions, in this case, the `method2` will have no condition for pair `method2` `method1`. For instance, the method `balance` does not have a commutativity condition with the method `deposit` but there is a condition for the

balance		deposit		transfer		withdraw	
balance	true	deposit	!(0 > balance + m2Args[2]) && !(0 > balance + m1Args[2])	deposit	!(0 > balance + m2Args[2])	deposit	!(0 > balance + m2Args[2])
transfer	true	balance	true	transfer	!(m2Args[3] > balance) && !(x2 > balance)	transfer	!(m2Args[3] > balance)
withdraw	true	transfer	!(m2Args[3] > balance)	withdraw	!(m2Args[2] > balance)	withdraw	!(m2Args[2] > balance) && !(m1Args[2] > balance)
		withdraw	!(m2Args[2] > balance)				

Table 3.5: Incomplete Commutativity map, with Java expressions for validation

balance		deposit		transfer		withdraw	
balance	true	deposit	!(0 > balance + m2Args[2]) && !(0 > balance + m1Args[2])	deposit	!(0 > balance + m2Args[2])	deposit	!(0 > balance + m2Args[2])
transfer	true	balance	true	transfer	!(m2Args[3] > balance) && !(x2 > balance)	transfer	!(m2Args[3] > balance)
withdraw	true	transfer	!(m2Args[3] > balance)	withdraw	!(m2Args[2] > balance)	withdraw	!(m2Args[2] > balance) && !(m1Args[2] > balance)
deposit	true	withdraw	!(m2Args[2] > balance)	balance	true	balance	true

Table 3.6: Final commutativity map, with Java expressions for validation

`deposit` method with the `balance` as it is possible to notice on Table 3.5.

Thus, it is necessary for each returned condition to define it with method's pair inverse execution order, and add it to the map if the map entry is empty. Said this, we obtain a complete map with no blank spaces. For instance, the entry (`balance deposit`) that in the Table 3.5 had no condition now has its condition in the final map, Table 3.6. This is important because the map is going to be consulted in runtime, and if all the entries are filled, there is no need to perform another search on the inverse order, and thus save time.

4

Implementation

This chapter describes the implementation of the developed solution stages in detail and discusses some relevant details. We will start by explaining the fundamentals of the bytecode analysis. After, we will describe the side effects extraction that involves primitive types. The next step is to explain how methods are extracted on objects and how method calls inside a method are resolved. The different types of dependencies will be explained, and also how the effects are stored for an object. The extraction of side effects ends with the invariants and method pre-conditions specification. The following section is the commutativity analysis. We will start by describing the changes made to the Servois. Then, we will specify the parsing from the `Effect` to SMT-Lib and explain the structure of the Yaml that Servois will consume. Finally, we will explain how the conditions are parsed from the SMT-Lib into a lambda function and the creation of the commutativity map.

The solution was developed in Java version 14. Graddle was used to support the use of external libraries and build the REPL project with all its submodules. The solution we have developed is inserted in one of the present REPL submodules, the REPL-Compiler.

4.1 Side Effects Extraction

4.1.1 Visitor and bytecode analysis

To extract expressions that define the method's execution effects, we utilize some previously developed work in the REPL project. The developed work as in its base the *ASM* [9] library, a framework with the objective of modifying existing classes, dynamically generating classes, or analyzing Java bytecode directly in binary form.

Our analysis starts by visiting the fields and method properties from the class.In a first phase, the code instructions are not visited. Instead, the fields and method nodes are visited. For each field node type, the first step is to analyze the annotations associated with the field; these annotations define the nature of the field and may define invariants, The second step is to extract the field type. In the end, this information is stored in a `Field` object that is added to the class complements.

Initially, for each method node, it is retrieved the method's type, i.e., if it is *void* or if it returns anything. Then, the method parameters, the parameter types, and the annotations are extracted and defined. In the end, all this information is stored in the method's information associated with the class information previously retrieved.

The second phase consists of performing a static analysis of the Java Bytecode instructions, identifying the instructions' actions and the instruction's actors, i.e., what resources are being affected.

So for the resources that are being affected, to easily identify them and store relevant information for each type of resource, those resources are created according to their type. There is a wide type of resources, but for the execution of the developed program, the most important were:

ConstResource this is a resource that represents a constant on the code. This resource contains the constant type and the value associated with it. In the initial REPL project, constants were ignored, and thus, this resource did not exist, but for commutativity analysis, the value of the constant is important to evaluation.

FieldResource this is a resource that represents a field on the code. This resource contains the parent Resource and a resource that identifies the object that guards it (a class where it is declared). Last, but not least, it contains the Field object that was described in the previous phase and contains information related to the types, nature, etc...

ObjectResource this is a resource that represents an object on the code. It is a very simple Resource that contains a map for complements. This complements map is useful because when resolving effect extraction on an object, it is possible to store in a complement the side effects that have affected the object.

ParameterResource this is a resource that represents a parameter on the code and identifies the method parameter index and its type.

BinaryOperationResult this is a resource that represents the result of a binary operation. This resource contains two operands and one operator.

StoreOpResult this is a resource that represents the result of adding or removing objects from an Array, a Map, or a List. It contains three resources. The first one is the structure that is being modified. The second is the key, which identifies the entry; the last is the value.

For the bytecode instructions, the visitor groups the different Java instructions in different operations according to the effect or parameters they need. The bytecode instructions to are grouped by the following operations:

Update is an operation that consists of instructions that affect a field and have an action of assignation. This operation is composed of the affected object (the target), the

affected field, the operator, which is the assignment (=), and an expression that will be the new value for the field.

Binary Operator is an operation that consists of decomposing the bytecode instructions that are composed of two operands and an operator. It is composed of two operands, which are resources, and the operator. Binary instructions might be seen in calculations and conditions, such as `if` instructions.

Unary Operator is an operation that decomposes the bytecode instructions and are composed of one operand and an operator array. Usually, it is possible to observe unary instructions in conditions where the second operand is 0. One example is the $x > 0$ expression, which results in a single instruction that takes only one parameter, which is the x .

Call is an operation that describes a method call (invoke virtual bytecode instruction), and it is composed of the target resource by the method name (the method that will be executed) and the parameters.

Method Return consists in identifying the resource that is returned by the method, and it is simply composed of the resource that will be returned.

Once we are only extracting side effects for replicated fields, the operations that we need to analyze are the ones that can change the field's value. Thus, from the above list, those operations are the **Update** and **Call**, so those are the operations we are analyzing to extract side effects. The **Method Return** operation is also analyzed because, when returning replicated values where consistency is required, this operation allows us to define later an auxiliary side effect to evaluate commutativity.

The flowchart from Figure 4.1, which occurs after the bytecode analysis, represents the first step of side effects extraction. During this first phase, only effects that affect replicated primitive fields are extracted. Thus, relying on the bytecode static analysis, we visit all the class methods and identify all the instructions inside each method. For each instruction, we analyze it and extract the corresponding effect. This process is represented in Figure 4.2, where it is possible to observe that four types of instructions are analyzed.

The **Update** and **Binary** instructions are used to extract the expressions which affect the replicated field, in which we describe all the processes in detail in Section 4.1.2, and the **Call** instruction is described in detail in Section 4.1.3. **Method Return** is used to generate effects on methods that return a value that must be consistent, and the process is described in detail in Section 3.1.2.

4.1.2 Effects over Primitive Types

Once primitive types are the most simple types and do not have dependencies, our solution starts by extracting side effects on primitive fields. Therefore, while analyzing the bytecode, the **Update** operation will be processed, and the side effects on primitive types

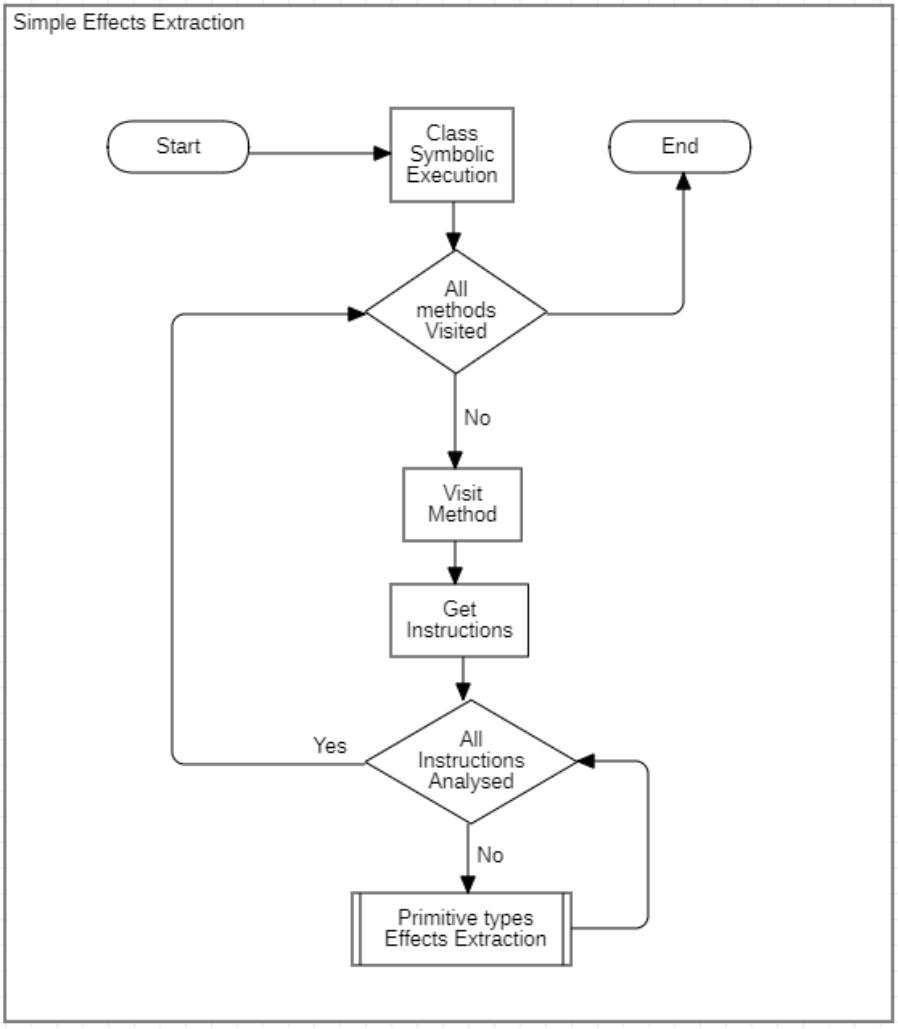

Figure 4.1: Simple Effect extraction static analysis and method visitor

```
update(ResourceReference object, String field,
       Operator operator, ResourceReference expr)
```

Listing 12: Update method signature

are be extracted. So, the **Update** operation changes the field's value by some expression. Thus, the `update` method, which identifies an update operation, has four parameters. The `object`(ResourceReference) represents a reference to the object which is being affected(Resource). As the name suggests, the `field` refers to the name of affected field. The `operator` represents the operation. Finally, `expr` represents the expression that will be the field's new value.

Thus the update method has the signature presented in Listing 12. To perform the side effect extraction, we start by retrieving the FieldResource associated with the field

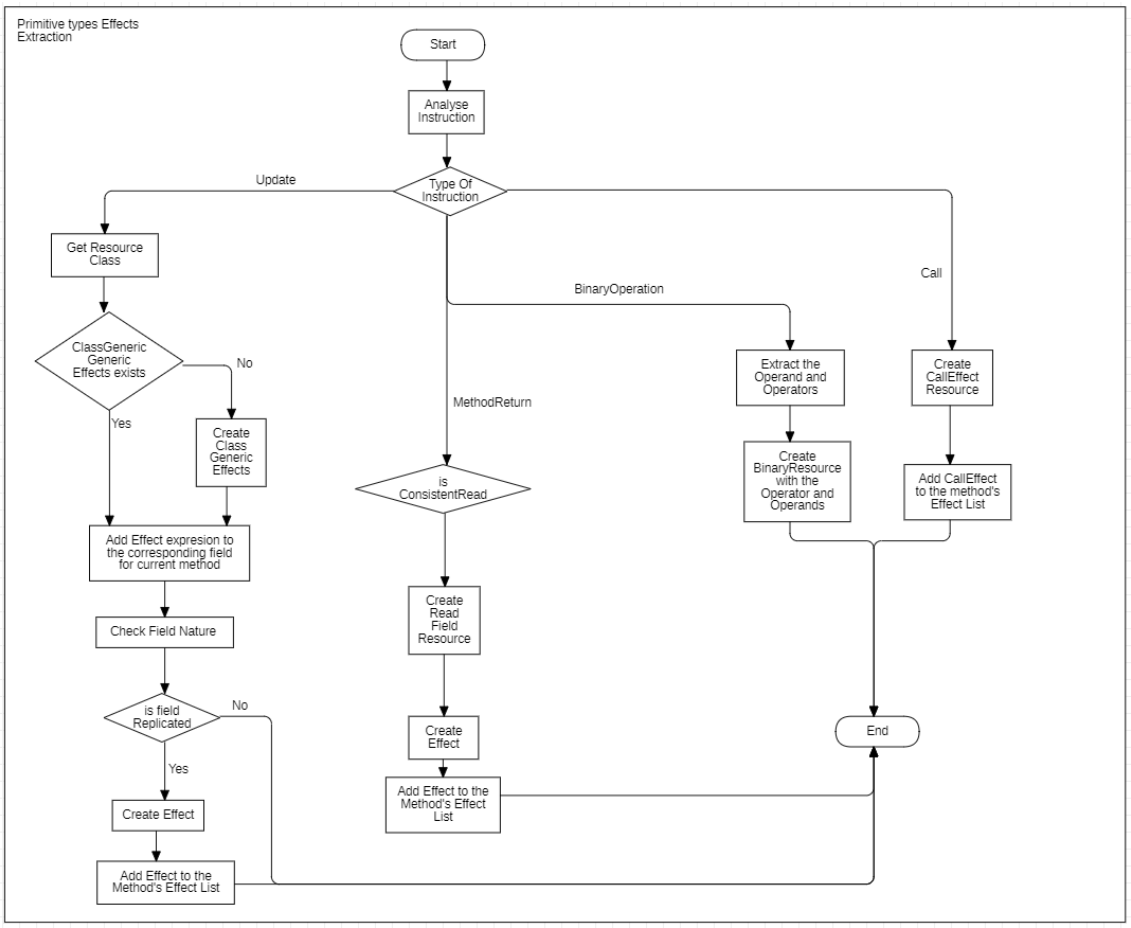

Figure 4.2: Primitive types Effect extraction (BinaryOperation,Call,MethodReturn,Update)

to update. As we only need to evaluate commutativity for replicated fields, if the FieldResource is not replicated, no side effects are extracted. On the other hand, if the value is replicated, an effect is created composed of the FieldResource that is being affected and the *expr*, which is the side effect.

An effect is implemented as a simple class (`Effect`) that contains two `Resources`, the left and the right. The left `Resource` identifies the affected object, and the right one identifies the effect expression. A small detail occurs when the expression is a constant and of type Boolean. The bytecode identifies this constant as zero or one instead of false or true, so it is necessary to convert the value to a Boolean value. Although we are still working on primitive types, the *expr* is not necessarily a constant, the *expr* can be a composition of operations. Thus we also process binary operations, which is an operation that contains two operands and an operator.

Although binary operations do not produce side effects, it is essential to define which operations are being performed and what resources are being affected because when side effect expressions are complex, i.e., they can be a composition of function calls and operations, it is possible to decompose them in binary operations. It is shown in Figure 4.3

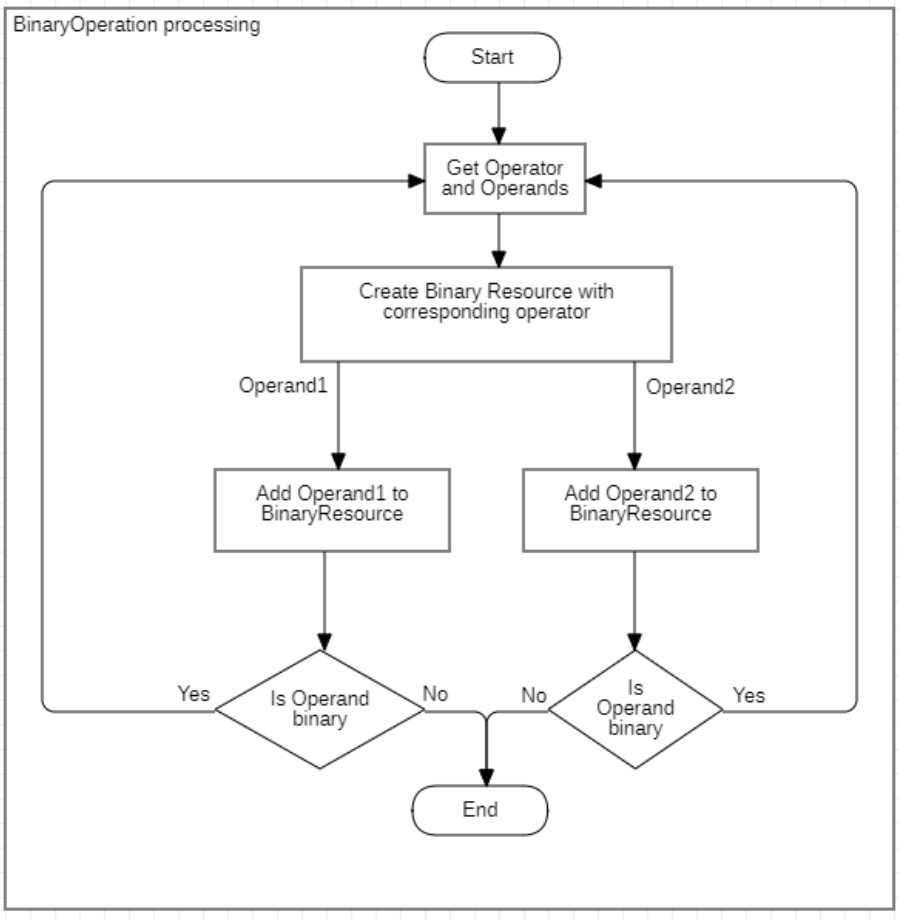

Figure 4.3: Binary Operations processing flow

the flow of processing a binary operation and the creation of the BinaryOperationResult Resource and, if there are more binary operations, the recursive creation of resources.

This reasoning allows the program to modularize the expressions, and once it works recursively, it is possible to have a large set of expressions. Thus, it is created a resource, of type BinaryOperationResult, constituted by the two operands and the operator. The operands are Resource, so it is possible to have another BinaryOperationResult inside the operands. Thus it is possible to encapsulate multiple BinaryOperationResults in a BinaryOperationResult.

So if x was a replicated field, for the following expression:

```
this.x = (this.x + 1) * 2
```

initially, the first binary operation would be resolved, which is $(this.x + 1)$, and it would be reflected in a BinaryOperationResult resource which would contain as operator the plus, and as operands would have the FieldResource *this.x* and the constant 1. Thus:

```
BinaryOperationResult aux = new BinaryOperationResult("+", this.x, 1);
```

then, the second binary operation, which is the multiplication of the previous BinaryOperationResult by 2, would result in:

```
BinaryOperationResult expr = new BinaryOperationResult("*", aux, 2);
```

Finally, the Update operation would define the side effect, which would be the last defined BinaryOperationResult.

```
Effect effect = new Effect(this.x, expr);
```

In the end, the method that is being analyzed is composed of some complements, and one of those complements is the EffectList which is a simple list which is used to store the method effects. So the extracted effects are stored in the EffectList complement.

Once we also need to extract effects on method calls and objects, it is necessary to store all method effects, although the fields may not be replicated. This has to be done because although the fields are not replicated, but the class object can be replicated. Thus we should store all effects in a complement for each method and variable.

So while extracting the effects on replicated fields, we analyze if the affected Resource is related to an Object (e.g.: if it is a field, it belongs to a class). If so, we create the GenericEffect complement if it was not created. The GenericEffect complement extends a Map that contains a Map(Map<String, Map<String, Resource»). The inner map contains the field which is being affected (String) and the expression that is being the new value (Resource). The outer map stores the inner map for each class method. This way, all the possible effects are stored, and when a method call invokes any of these methods, it is possible to consult the map and retrieve the effect.

As we analyze a custom class, to support the object effect extraction, if this class is used as a replicated object, it is necessary to declare in Servois its preamble, the constructor, etc..(section 4.2.1.4). Thus during the first effect extraction on the class, the ServoisMapping complement is created. This complement stores the class fields and their types in the SMT-Lib syntax, the Servois class declaration, the Servois constructor, and the preamble, described in more detail in section 4.2.1.4.

With these two complements, it is possible to define effects on objects using datatypes, although it is necessary to specify the effects, which is described in section 4.1.3.

4.1.3 Objects hierarchy extraction

After extracting the side effects on primitive types, the next step is to extract side effects on objects or in method calls.

The analysis of a method's implementation simply collects method calls, postponing its resolution to a subsequent inter procedural analysis stage. The CallEffect object contains information relative to the method call, such as which object and method is calling and which method and class are being called. So, the next step is to resolve these CallEffects that still need to be resolved.

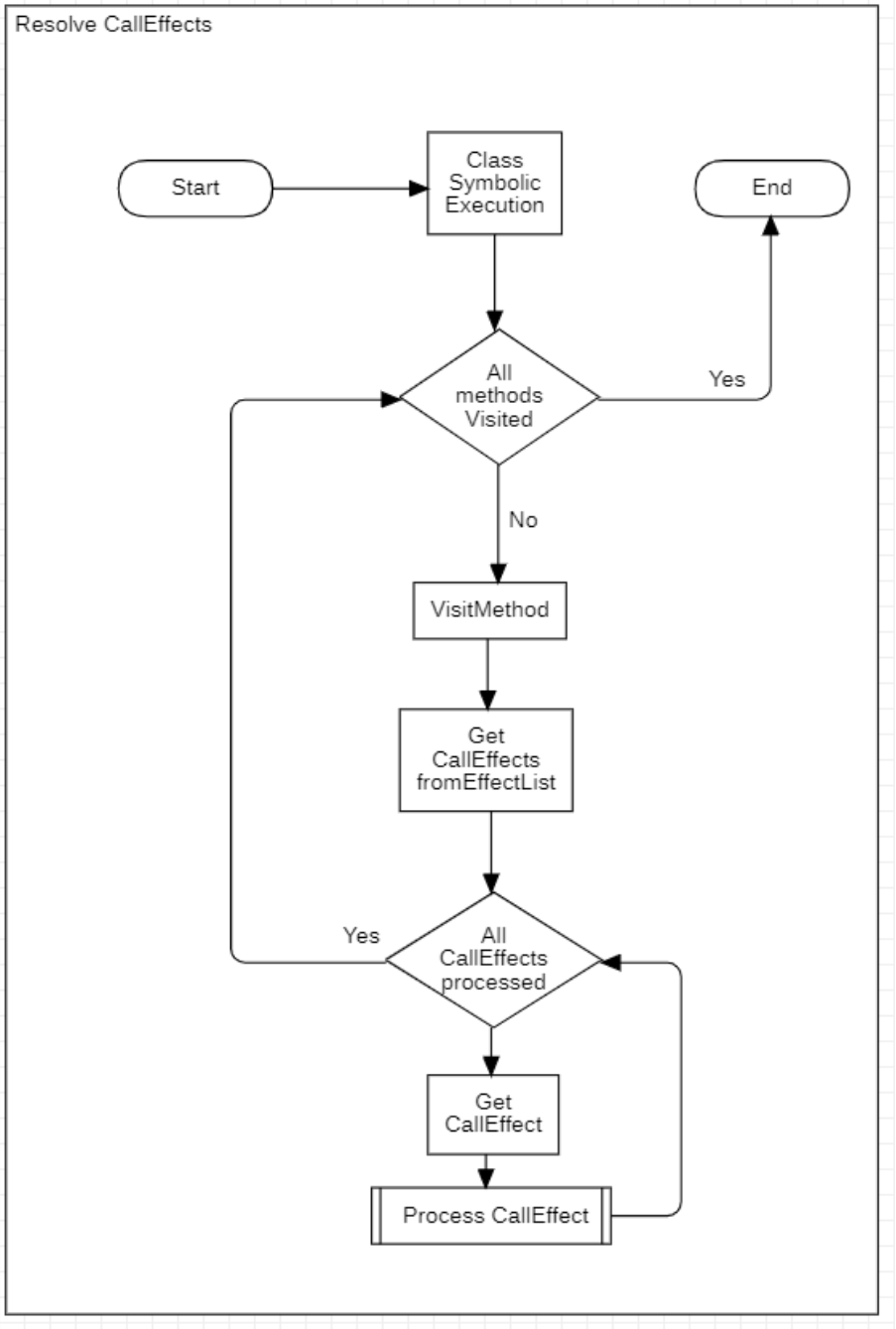

Figure 4.4: Object effect extraction static analysis and method visitor from CallEffects

To that end, we visit every method from the class and verifying if the EffectList contains any object of type CallEffect. If the method's effect list does not contain any CallEffect, the method is marked as resolved and advances to the next one. Otherwise, every CallEffect in the effect's list must be resolved. This process is illustrated in Figure 4.4. Four types of CallEffects have been identified.

The first one is just a simple method call, where the methods belong to the same object. One example of this CallEffect is the method transfer which calls the withdraw (Account class Listing 4 Chapter 3). The second is a call to an object that is not replicated, but it is necessary to resolve the method call because, although the object is not replicated, one of its fields may be. The third is the call to the constructor of an object. Although the object may not be replicated, it is necessary to evaluate the new object because later in the program, this object may be assigned to a replicated field. Finally, the fourth is the call to a method of an object that is replicated.

For every method call, an object of type `Method` stores the method's code dependencies graph, i.e. that types it depends upon. So, in order to have the side effect's information for all of such types, these dependencies have to be resolved first, before resolving each method call.

For the first type of CallEffect mentioned above, we start by resolving the dependencies. As the target method is in the same class, the target method is visited. Following the diagram in Figure 4.5, once it belongs to the same class, the effects for the target method have already been extracted. Thus, we extract the target method effects. Then we process the target method effect list following the procedure depicted in Figure 4.6.

Once, in this case, the target method has no method calls, it is not necessary to resolve them, but it has effects. So each Effect from the target method's EffectList is added to the calling method EffectList.

There is a small detail which has to be performed in all method calls. When adding the target method effects to the calling method, the effect expression has to be verified because if there is a parameter, that parameter has to be mapped to the value that the calling method is passing. In the code presented in Listing 13, after resolving the method call `multiply` performed in method `double`, the returned Effect would be $(= \ x \ (* \ x \ y))$, although the correct Effect for `double` should be $(= \ x \ (* \ x \ 2))$. Thus, parameter mapping is performed before adding the Effect from the target method to the calling method. This procedure is applied to the parameters and binary operations because one of the binary operands might be a Parameter Resource.

Returning to the method dependencies, it is necessary to analyze the dependent class if the dependency is not from the same class. If the class has yet to be visited, we will proceed to the Java bytecode analyses as previously. Again the visitor analyses the fields and methods. A static analysis analyses the bytecode, then effects on primitive types and method calls are extracted, as presented in Figure 4.5 on the *Different Classpath* branch on the left. The next step is to resolve the method calls, and this flow will be executed recursively until no more dependencies exist.

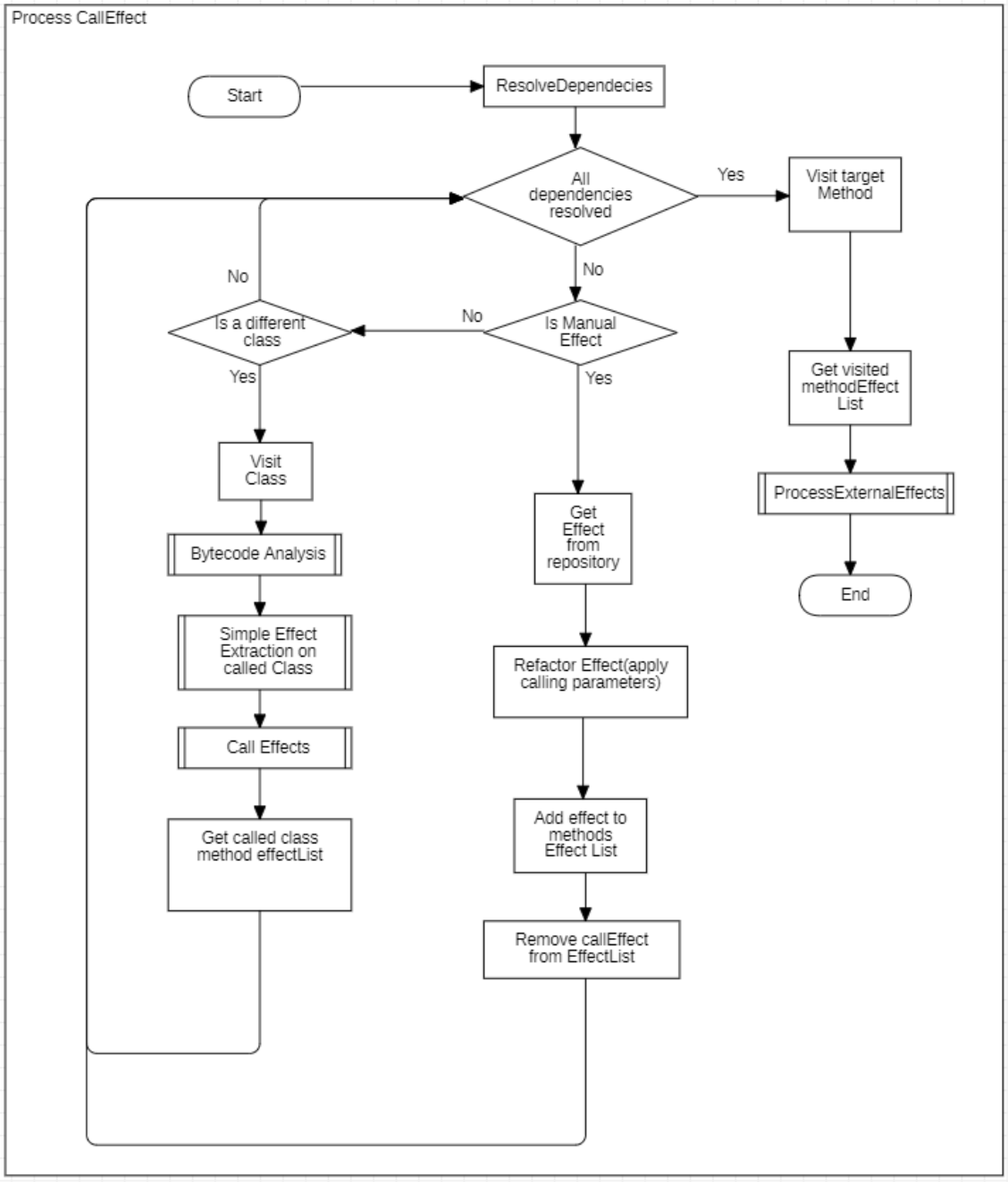

Figure 4.5: CallEfect process, verification of callEffect type, if it is from a different class, it is from a class defined in the manual effects or from the same class as the calling method

```
@Repl int x;

void multiply(int y) { x= x * y; }

void double(){ multiply(2); }
```

Listing 13: Example of why the parameter must be mapped

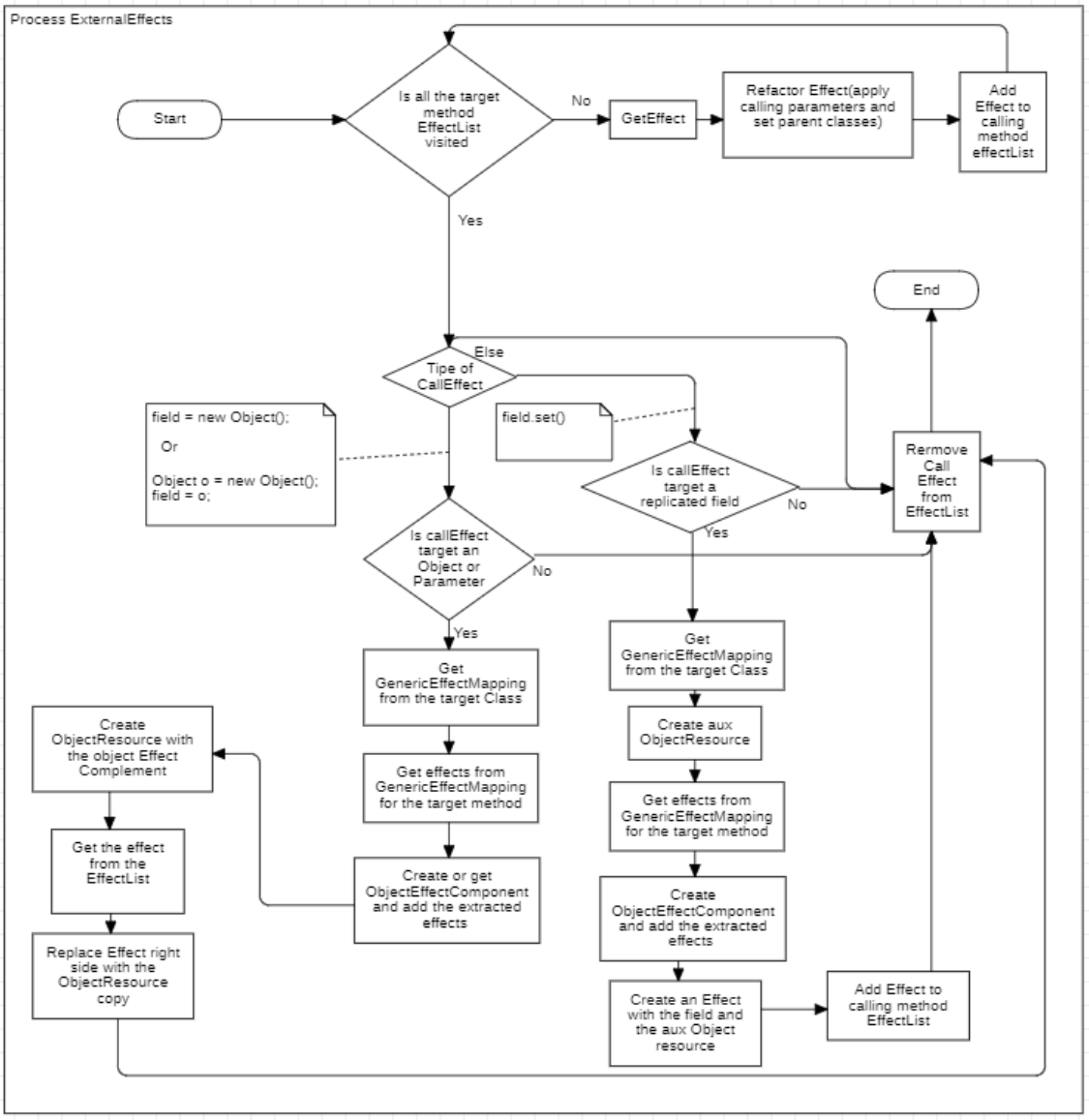

Figure 4.6: Process callEffect from an External class (class different from the calling class)

While analyzing the classes recursively and extracting the side effects for each of the class methods, only the replicated fields produce side effects. Thus, those replicated fields will have an effect on the class that we were analyzing first. To define an effect for those fields, their names are concatenated with their parent's field name. For example, in the classes presented in Listing 14, the side effect produced for the class Z is $(= \ this.a \ (+ \ this.a \ 1))$, for the class Y is, $(= \ this.z.a \ (+ \ this.z.a \ 1))$, and the effect produced for class X is $(= \ this.y.z.a \ (+ \ this.y.z.a \ 1))$.

After extracting the side effects on the subclasses, as demonstrated in the previous example, it is still necessary to resolve the initial `callEffect`. If the `callEffect` does not involve a replicated field, the `callEffect` is removed. Otherwise, if the `callEffect` involves replicated object is necessary to extract the effects produced on the object(not only on the replicated fields from the object). This introduces the the third and fourth

```
class X {
    Y y;

    void incY() { y.incZ(); }
}

class Y {
    Z z;

    public void incZ() { z.inc(); }
}

class Z {
    @Repl int a;

    public void inc() { a++; }
}
```

Listing 14: Example that requires recursive analysis

cases of callEffect types mentioned above.

So, when a method call has an Object resource as a target, we need to extract the effect information generated for that object. There are two types of operations on objects where the effects are extracted, the assignment of an object field to an object variable or a method call on the object field, which produces an effect, for instance a set method.

Thus, knowing all possible side effects for the called method is necessary even if the fields are not replicated. During the primitive field extraction phase, we also extract the side effects for non-replicated fields, and all the side effects are stored in the method information. Thus, we have all the side effects for all the fields associated with the method.

Following Figure 4.6, after visiting the `EffectsList` from the target method, we have to identify which type of `callEfect` it is.

We have two: an assignation to a field Object or a method call directly on a field Object.

For the first one, when a call effect has an Object as the target, we consult the method information of the called method and get all the method side effects(not only the replicated). Then, a structure is associated with the object where all these effects are stored. Then, we verify if there is any effect that has a field and the processed object. If an effect exists, the right resource is updated with the object(containing the structure with all the side effects).

In the other case, where a method call invokes a field directly, we also consult the method information and extract all the effects. It is created an auxiliary Object, and the side effects extracted are associated. In the end, as we have a `callEfect` directly on the field, we do not have the correspondent effect in the `EffectsList`. Thus, we

create an `Effect` involving the called field and the auxiliary object. In the end, removing the `CallEffect` from the `EffectsList` is necessary once we have already added the correspondent effects.

4.1.4 Manual Effects on Data Structures

While resolving the dependencies, before checking if the target class is different from the calling class, it is analyzed if the target class is in the manual Effects repository referred to in Section 3.2. If the class is not present in the repository, the analysis proceeds to the analysis defined in section 4.1.3. Otherwise, as demonstrated in Figure 4.5, the method's manual effects are extracted, and then the parameters are parsed to the method call's parameters. Finally, the new effects are stored in the method's Effectlist, and the CallEffect is removed.

4.2 Commutativity Analysis

After analyzing the code and extracting the side effects and conditions, the next step is to evaluate commutativity. To evaluate commutativity, the Servois auxiliary tool has been used, although to support some expressions, some changes had to be made to the original Servois program. During this subsection, we will present the changes performed on the Servois program, and we will show how the expressions and conditions are parsed to SMT-Lib, the input language supported by Servois, and how the input file is created. Additionally, we will explain how the outputs from Servois are generated and clarify the parse from the Servois output to java biFunctions.

4.2.1 Servois

Servois takes as input an YAML file which is has the following properties: State, Methods, Predicates, Preamble and State Equals.

State corresponds to the declaration of the fields that Servois is going to analyze. In the State, we should define the field name and its type.

Methods corresponds to the method's declaration, where the side effects expressions and the conditions/invariants are defined. Additionally, the terms are also declared, and they will be the main component to generate commutativity conditions.

Predicates define the supported operations and types the operation involves. A correct predicate declaration is fundamental because, without the predicates, the term will not be compared, and thus, no conditions are generated.

Preamble is an SMT-Lib declaration that is not mandatory, but it allows to define SMT rules and additional logic to the solver.

State Equals defines the expression which will be used to verify if two states are equal and thus identify if two operations commute.

In the first phase, by analyzing the Servois examples manually, we tried to produce some inputs according to the classes we had defined that produced the desired results. As the Servois executable output is on Latex, one of the first changes was to define an output syntax that our program could parse to lambda functions. Additionally, after running some tests on primitive operations, we found that Servois did not simultaneously support expressions containing additions and multiplications which meant that we had to make some changes to Servois.

4.2.1.1 Modifications to Servois

The first change on the Servois tool was to return an output file with a syntax that could be easy to work with and easily interpreted by the human eye instead of the latex code. The syntax defined for the output can be split into three columns, the first the method1 and the second the method2. It means that method1 might commute with method2 if the condition is true. The condition is defined in the third column, and the syntax of the conditions is the SMT-Lib syntax.

After some research, we found out that the failure in evaluating expressions with addition and multiplication was caused by the SMT-solver used by Servois, which was the CVC4, and it was unable to identify if the expression was solved, neither proving that the expressions were impossible. Thus, in order to resolve this problem, we have implemented the Z3 solver in Servois. While CVC4 could be executed, and the commands were interactive. Thus the Servois called a system subroutine for CVC4 and passed the commands as a String with the standard input pipe. The Z3 did not support the interactive execution, but Z3 has a Python API that would require many changes in the Servois core. Happily, Z3 also supports an smt-lib file, so we write the input in a file, and Z3 consumes it. Although this solution might not be the most efficient because writes on disk are slow, this was the least time-consuming for the project's development.

4.2.1.2 YAML Creation

As in the previous phases of the program execution, the class and the methods are revisited in this phase.

The first step to create the yaml file that Servois expected was to create a POJO class as demonstrated in Figure 4.7. In this case, two classes were created the `ServoisYamlformat` class, which contained the main structure, and the `MethodFormat` class, which contained the structure of the method and was created while visiting the method. Using these POJO classes with the help of the `YAMLFactory` and `ObjectMapper` classes, it was only necessary to pass the `ServoisYamlformat` object, and a YAML file with all the fields from the POJO classes was generated.

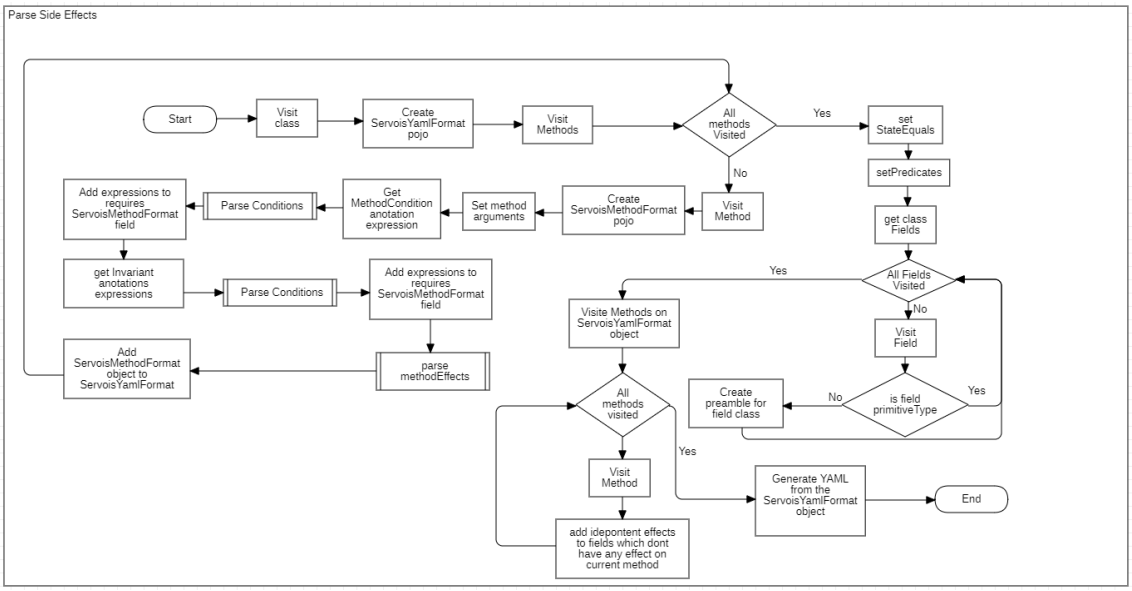

Figure 4.7: Parse side effects into SMT and creation of the YAML Servois input file

So while visiting a class, the `ServoisYamlformat` is created. The `ServoisYamlformat` is composed of the properties specified in section 4.2.1 (Methods, Name, Predicates, Preamble, State, and StateEquals). The POJO fields implementation is presented in listing 15. Initially, only the name field is set when the class is created. The rest of the fields are only set after visiting all the methods because the state is only set for fields with effects, such as the replicated fields that produced effects. The methods field is a set of `ServoisMethodFormat`, which is another POJO that contains the method properties. It is composed of the arguments, and the effects, which are represented by the ensures field. The requires, which represents the post-conditions, the rets, which represents the method return, and the terms. Additionally, there are the presentFields set, which is used as an auxiliary to define the `ServoisYamlformat` State field. The POJO for the `ServoisMethodFormat` is also presented in listing 15.

So after the creation of the `ServoisMethodFormat` POJO, it is necessary to set its fields, so as the figure 4.7 suggests, the first `ServoisMethodFormat` field to be defined is the arguments which are obtained directly from the Method object. After argument definitions, the following step defines the required field, which analyses the invariants and the method conditions.

4.2.1.3 Conditions parse

The field requires represents the method post-conditions. Thus the complement, which contains the invariants and the method conditions, is analyzed, and the conditions are extracted. As the expressions are strings to parse them to the SMT-LIB format, we have recured to JAVACC [24] to convert them to smt-lib. These expressions have a syntax as defined in Section 3.1.1. Using regular expressions, we analyze the string that is obtained

```
public class ServoisYAMLFormat {

    private String name;
    private Set<ServoisVariable> state;
    private List<ServoisMethodFormat> methods;
    private Set<ServoisPredicates> predicates;
    private String stateEquals;
    private String preamble;

    ...
}

public class ServoisMethodFormat {

    private String name;
    private List<ServoisVariable> args;
    private List<ServoisVariable> rets;
    private String ensures;
    private String requires;
    private Map<String, Set<String>> terms;
    private Set<String> presentFields;

    ...
}

public class ServoisVariable {

    private String name;
    private String type;
    ...
}
```

Listing 15: YAML POJO classes

from the method condition or invariant complement, and according to the tokens that are present in the expression, the expression takes different paths.

We have defined the following Tokens (Listing. 16) to define the identifiers(variables), comparators, and boolean operators. The **NOT** token is represented by "!"and represents the negation of the following expression. The other Boolean operators are the **AND** and **OR**, which represent the conjunction or disjunction, respectively. For **COMPARATOR** token, we have the equals, the bigger than and lower than, which are used to compare to expressions. Additionally, we have the **NOTE**, the **GREATERE**a, and the **LOWERE**, which represent the not equals, greater than or equal, and lower than or equal, respectively. These tokens are also comparators, but they could not be in the same token as the greater, lower, and equals because they used characters from these tokens. We also have the

```
TOKEN: { "(" | ")" | "" |<NUM: (["0"-"9"])+>}

TOKEN: {<COMPARATOR:["=",">","<"]> | <GREATERE:">="> | <LOWERE:"<="> }
TOKEN: {<NOT:"!"> | <NOTE:"!=">}
TOKEN: {<OPERATOR:["+","-","*","/"]>}
TOKEN: {<AND:"&&"> | <OR:"||">}
TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER: ["_","a"-"z","A"-"Z"] >
|
  < #DIGIT: ["0"-"9"] >
}
```

Listing 16: JAVACC TOKENS parsing from java to SMT-lib

OPERATOR token, which represents the operators for a mathematical expression, and the **IDENTIFIER** token, which represents the variables and is composed of letters and digits.

An overview of the parsing logic is described in figure 4.8, where the first step is to check for the **NOT** Token. If the token is absent, we analyze the rest of the expression. In contrast, if the token is present, the SMT-lib negation is written, and we proceed with the remaining expression analysis. The resulting expression is (*not expr*1) , and we proceed to analyze the remaining expression, which in this case would be the *expr*1 .

Then we look for the **AND** and **OR** tokens to slice the expression in two. If the token is present, we define the **AND** or **OR** SMT-Lib representation, and we proceed to analyze the two expressions that result from the separation. The resulting expression for the **AND** would be (*and* (*expr*1) (*expr*2)) , and for the **OR** would be (*or* (*expr*1) (*expr*2)) .

If none of these tokens is present, the expression is not divided in two, and the analysis proceeds with the original expression. The subsequent analysis is the binary operators' analysis, where we look for the **OPERATOR**, **COMPARATOR**, **GREATERE**, **LOWERE**, and **NOTE**.

Then we look for method calls, method calls are identified with an identifier(the variable or the class), followed by a dot ("."), then another identifier(the target method), and then multiple identifiers may represent the parameters inside the parenthesis. The regular expression is represented as **<IDENTIFIER> "."<IDENTIFIER> "("expr1 ")"** .

The next step is to evaluate if the expression is surrounded by parenthesis **(expr1)**. If the expression is surrounded by parenthesis, the parenthesis are removed, and the remnant expression is analyzed starting from the first step described above. Otherwise, it proceeds to the final step.

Finally, we look for single Identifiers or Numbers, which are the primary operands. This analysis is recursive, and thus when an expression is checked for a TOKEN, it starts

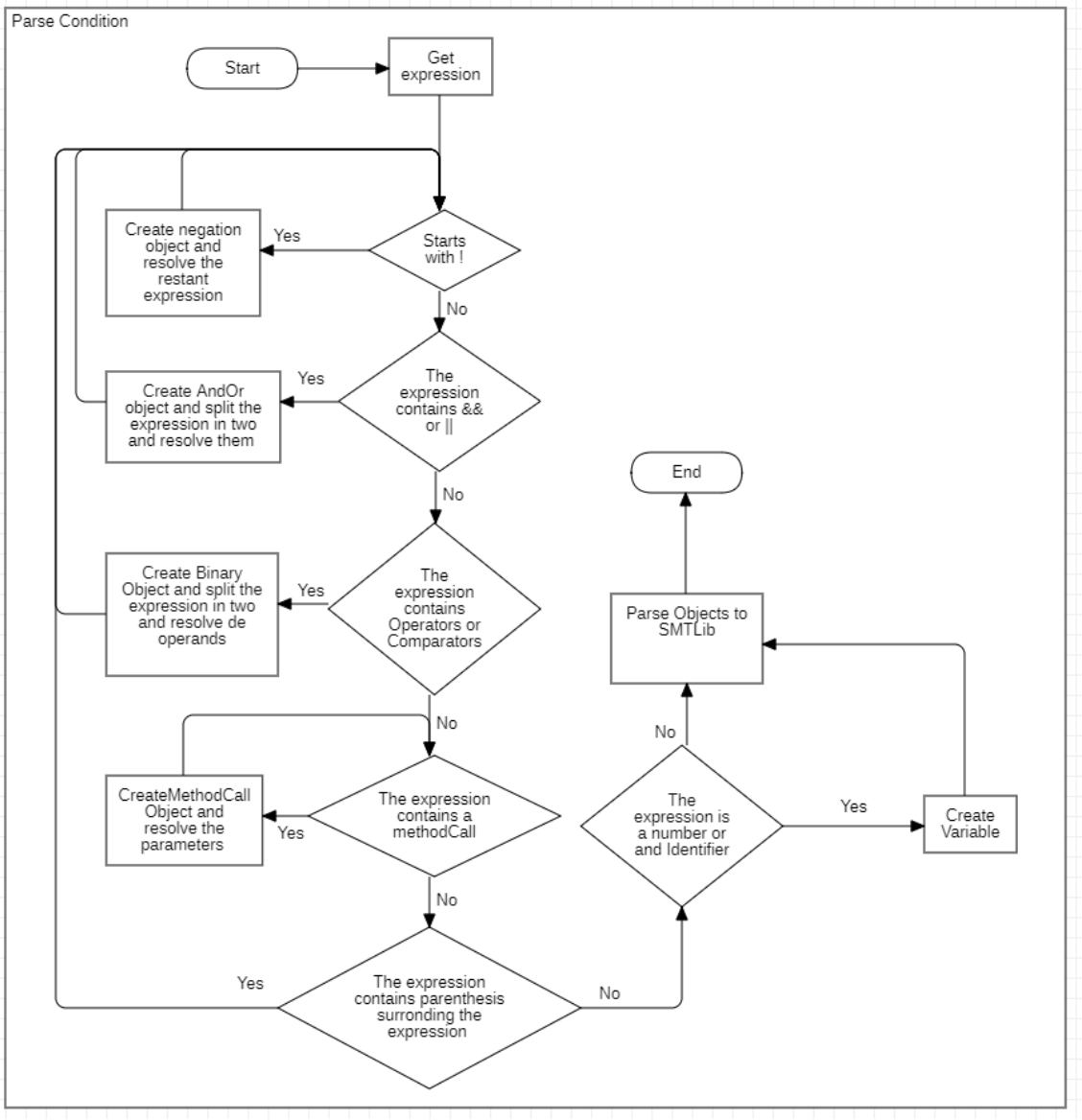

Figure 4.8: Parse Invariant/method condition string to SMT-Lib using JavaCC

the process from the beginning with a new expression. With this definition, we are able to parse the invariants and method conditions that are specified as a String in the manner of Java to SMT-lib syntax. Thus after defining the method's postconditions in SMT-Lib, we advance to the effects specification.

4.2.1.4 Side effects Parsing

The process of parsing the effects, extracting terms, and defining predicates is demonstrated in figure 4.9. Thus we start by getting the effects from the `EffectsList`, and each affected Field. The affected fields are stored in the `ServoisMethodFormat` presentFields to identify later which fields have been affected by each method.

As defined before, the effects consist of two resources, the left, which is the replicated

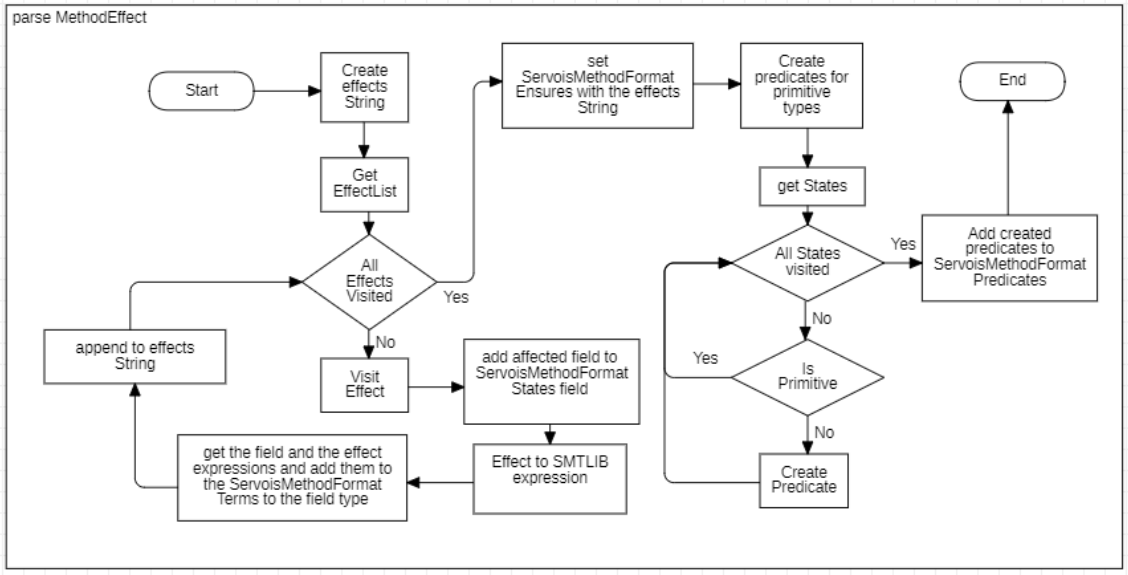

Figure 4.9: Creation of the Servois input Yaml file, predicates and states

Field, and the Right resource, which can be any of the resources specified in section 4.1.1, and can be a more complex expression that can be decomposed into binary operations.

The Field Resource is represented in SMT-Lib with just the field name. Additionally, if the Field has a parent, then the parent field name is concatenated with the field name. This process may be recursive once a parent field may have another parent. So all the parent relations are expressed. The ConstResource expression parsed to smt-lib is simply the constant value that is defined in the code. To parse the parameterResource to SMT-lib, the parameter index is only extracted, and the index is concatenated to the String "param". The BinaryOperationResult, used to express a more complex extraction, is defined in the smt-lib syntax starting with the operator and then the two operands, which results in (*operator operand*1 *operand*2) . The StoreOpResult is parsed to smt-lib following the logic of the SMT theory of arrays store. It is defined by the **store** keyword followed by the structure name, index, and value. It would result in (*store structure key value*) The final resource that is parsed is the ObjectResource. As defined before, the object resource stores all the effects for each of the object's fields, and thus it is necessary to define an expression where the effects are defined for each of the field objects. So the adopted strategy was to express the object effects as tuples and adopt the SMT datatypes. With this solution, each tuple field will have the effect of the corresponding object field. According to the smt-lib, the datatypes must be declared, i.e., it is necessary to define the datatype, how it is composed, the names of the fields, and the types. So we created a complement for each class containing the SMT-lib additional information necessary to define a class as a datatype.

The complement contains an expression to define the object as a datatype called the preamble. The complement preamble is added to the yaml file preamble to define the

datatype. The preamble consists of a string with the following format:

$$(declare-datatypes\ (\ (<classname><num\_of\_fields>)\ )$$
$$(\ (\ par\ (<field\_values>)\ (\ (<constructor>)\ )\ )\ )\ )$$

The *classname* is the name of how we desire to identify the datatype. In the project, we use the class name of the object. The *num_of_fields* is the number of the class's fields. The fields **super** and **this** are discarded, so they are not used for counting. Then the *field_values* is simply an identifier of the fields. Finally, we have the <*constructor*>, which has a declaration(the *classname* in lowercase), the fields names and the associated *field_values*.

$$(\ (<classname\_lower>\ (<field\_name_1><identifier_1>)(<field\_name_2><identifier_2>)...\ )$$

Then to define an object of the *classname* type it is necessary to declare it. To declare it, it takes the *classname* and the object fields associated types.

$$(<classname><field\_type_1><field\_type_2>...)$$

The last auxiliar expression we need is the class constructor, this constructor is used to assign values to the dataype. It is requires the *classname* in lowercase and the *fieldnames*.

$$(<classname><field\_name_1><field\_name_2>\ ...)$$

During the side effect parsing the *fieldnames* are substituted by the corresponding field effect and thus, a new value to the datatype .

The *constructor* is also stored in the complement because it will be used when defining an object effect. Additionally, it is stored in the complement of the SMT datatype field declaration, which is used to define the field and contains the datatype and the field types.

For example, for this Pair class listing 17, which is a tuple of Integers, we would have the complement with the following characteristics:

Preamble:

$$(declare-datatypes\ (\ (\ Pair\ 2)\ )\ (\ (\ par\ (\ T1\ T2)$$
$$(\ (\ pair\ (first\ T1)\ (second\ T2)\ )\ )\ )\ )\ )$$

Declaration:

$$(\ Pair\ Int\ Int\ )$$

Constructor:

$$(\ pair\ first\ second\ )$$

After this specification, we are able to express the effects of the object Resource. To specify the effect on the object, the constructor is used. If all fields of the object produce

```
    public class Pair {

    private int first;
    private int second;

    Pair(int first, int second){
    this.first = first;
    this.second = second;
    }

    public int getFirst(){
        return first;
    }

    public int getSecond(){
        return second;
    }

    public void setFirst(int first){
        this.first = first;
    }

    public void setSecond(int second){
        this.second = second;
    }
}
```

Listing 17: Pair class ith int

an effect, then the constructor fields are the object effects. For instance, the method Pair of the Pair class, listing 17, produces side effects on both fields, so the object resource has in its effects map, the field first with the effect the parameter0 and the field second the parameter1. Thus the SMT expression representing this effect would be $(pair\ param0\ param1)$. On the other hand, the methods set that only affects one of the fields also uses the constructor syntax but for the fields that do not have side effects it is used the value that is assigned to the object at that time. To define that value, we make a get call to the datatype asking for the value of the required field. In SMT, it results in the following, $(field\ datatype)$. So the SMT-lib expression for the method setFirst of the class 17 results in $(pair\ param0\ (second\ pair))$. The method setSecond results in $(pair\ (first\ pair)\ param0)$. The "pair"is the field's name in the yaml representing the class Pair object.

With the object resource effects parse, we finalize the side-effect parse to SMT-lib. The next step after effect parse to SMTlib is the Terms extraction.

```
Term:
   Int:
   - field1
   - (+ field1 (* param0 2))
   - param0
   - 2
```

4.2.1.5 Terms

To specify the terms of the method, we analyze each side effect. As terms need the type, the first step is to identify the type of side effect. As the effect has the right and left sides, the left resource contains the field resource, so it is simple to extract the field type. Thus, it is created as a term for the given type with the field. For the right resource of the effect, as the resource can be a set of another resource, the expression type is the same as the field (Left Resource) because, as the left side is going to be assigned to the right resource, they have to be of the same type. Additionally, as the right resource may be a set of expressions, it is necessary to analyze it and extract the subexpressions and the correspondent types. The expressions are analyzed recursively till reaching the simplest expressions. It means that a complex expression, defined as a binary Operation, is decomposed into small expressions, and when these expressions are parameters or constants, the type is extracted.

For instance this simple example, $(= \ field1 \ (+ \ field1 \ (* \ param0 \ 2)))$, where the $field1$ is the left resource, and $(+ \ field1 \ (* \ param0 \ 2))$ is the right resource. Assuming that $field1$ is an int, then $(+ \ field1 \ (* \ param0 \ 2))$ is also an int. So, while analyzing the right resource, the resource is divided into two expressions $field1$ and $(* \ param0 \ 2)$. As $field1$ is a simple expression, we can extract its type. As $(* \ param0 \ 2)$ is a complex expression, then it is necessary to divide again. Two simple expressions are obtained from the new division: $param0$ and 2. Thus it is possible to extract their type. Assuming that the $param0$ is of type `int`, the resulting term would be:

4.2.1.6 stateEquals and predicates generation

Thus, after processing a method, the `ServoisMethodFormat` for the correspondent processed method is added to the `ServoisYamlFormat`. In the end, it is necessary to define the statesEquals YAML field, which represents the condition of equality for the commutativity evaluation, and the definition of predicates.

The stateEquals condition is quickly built by expressing the equality predicate for each state (class replicated fields). Thus for each state, we define the equality predicate, $(= \ state1_1 \ state1_2)$. In the end, all the equality predicates are concatenated, $(and \ (= \ state1_1 \ state1_2) \ (= \ state2_1 \ state2_2) \ ...)$.

The predicates need a little bit more work. It is necessary to specify the name of the predicate and the type of terms it evaluates. For instance, if we want to enable the

```
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
   type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
```

conditions generations involving integers, it is necessary to specify the possible operators that can be applied to an integer. Thus for creating conditions to evaluate commutativity on integers, it would be necessary to define the equals, the bigger than, and the lower than, like in the following listing.

With this specification, Servois is enabled to generate predicates and thus create commutativity conditions. Notice that a condition can be a set of predicates connected by boolean connectors.

In our program execution, the predicates are created after defining the terms for all methods. The predicates generation starts by analyzing the replicated field types, and for each type, it generates the equality predicate. In addition, the inequality, bigger and lower, are added for integers. After predicates generation, the POJO ServoisYAMLFormat class is completed. There are two more steps before having the YAML ready for execution.

Each of the class replicated fields is checked if the field is a primitive type. If not, it is necessary to define the preamble for the correspondent object type as defined in 4.2.1.4. After the preamble is defined for all complex objects, the methods(`ServoisMethodFormat`) are revisited and idempotent effects are added for fields that are present in the states but have no effect on that method. This is a process required by Servois, where all the states must affect every method. As we need an effect that does not change the state of the field, we create a dummy effect which is $(= \ field1_newfield1)$. This way, we specify that the field is going to have the previous value.

After all these processes, the YAML file is ready. Thus using the auxiliary classes

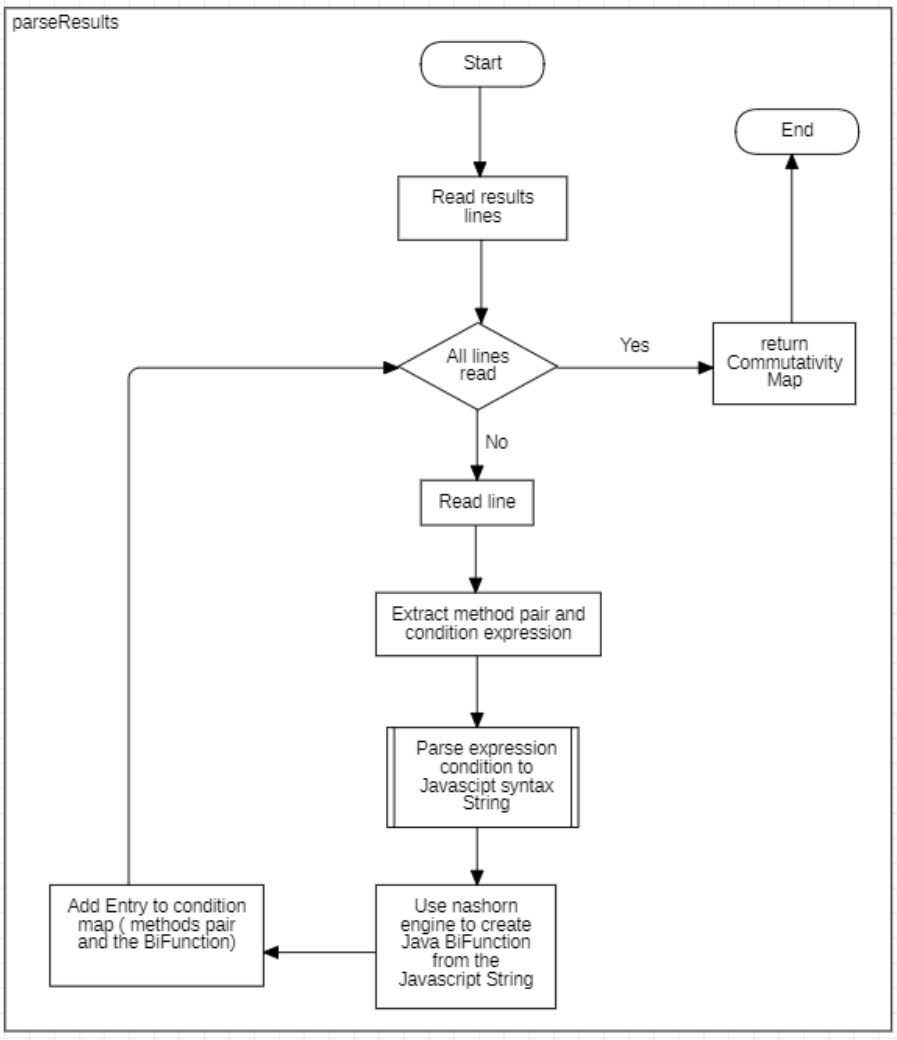

Figure 4.10: Parse Servois results from SMT-Lib string into Java BiFunction

`YAMLFactory` and `ObjectMapper`, the YAML file is generated and saved in the Servois directory. For the Servois execution, a system call is created, which passes the commands for the execution of the custom Servois python program and the input YAML file name.

4.2.2 Verification Map

After the Servois execution, the results file is saved in a *.txt*. Each line represents a commutativity pair and the commutativity condition associated with the pair so that it can be decomposed into three columns, the method1 name, the method2 name, and the commutativity condition. As shown in figure 4.10, we start by reading line by line, and for each line, we extract the commutativity condition. The commutativity condition is defined as SMT-Lib syntax. Thus it is necessary to convert it to a lambda function. First, the condition String is parsed from SMT-Lib syntax to JavaScript syntax, and then the String is parsed to a lambda function. In the end, the lambda function (BiFunction) is added to the commutativity map. To parse the SMT-Lib syntax to JavaScript syntax, the

```
TOKEN: { "(" | ")" | " " }
TOKEN: {<TRUE:"true"> | <FALSE:"false"> | <FALSE_UNKOWN:"false_un">}
TOKEN: {<AND:"and"> | <OR:"or"> | <NOT:"not">}
TOKEN: {<EQ:"="> | <BIG:">"> | <LOW:"<"> | <BIGE:">="> | <LOWE:"<="> }
TOKEN: {<SUM:"+"> | <LESS:"-"> | <MULT:"*"> | <DIV:"/"> }
TOKEN:{
         <NUM: (<DIGIT>)+ (".")? (<DIGIT>)*>
}
TOKEN:{
         < PARAMETERS: <LETTER> (<DIGIT>)+ >
       |
         < #LETTER: ["x","y"] >
       |
         < #DIGIT: ["0"-"9"] >
}

TOKEN : {
    < OTHER : <OTH> (<IDENTIFIER>)* >
    |
    < #OTH : "OTHER."> 
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <ALPHABET> (<ALPHABET>|<DIGIT>)* >
          |
          <#ALPHABET: ["A"-"Z","a"-"z"] >
}
```

Listing 18: Javacc TOKENS parsing from SMT-lib to Java

JAVACC [24] was used again.

4.2.2.1 Parse smtExpression to Java

First the tokens were defined (listing 18), these token allow us to define the variables, the operations and in some cases the commutativity property.

The process of parsing from SMT-Lib to javascript is described in figure 4.11. The first step is to check for the **TRUE**, **FALSE**, **FALSE_UNKNOWN**. These tokens express the base case where independent of the state or parameters, and the commutativity evaluation does not change. In this case, if the token **TRUE** is present, it means that the method's pair always commute. The other two tokens express that the methods never commute, and the **FALSE_UNKNOWN** additionally expresses that Servois could not get to a conclusion. Thus, to ensure safety, it is assumed that methods never commute.

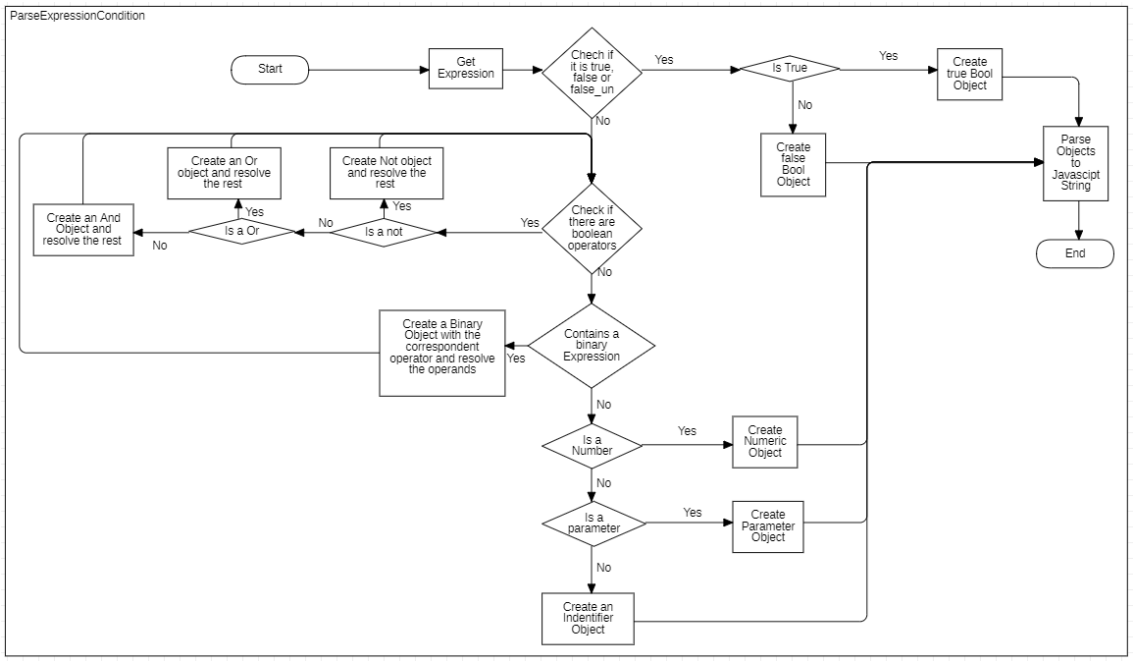

Figure 4.11: Parse the SMT-Lib expression to JavaScript syntax string using JavaCC

Keep in mind that this conclusion is only for this method's pair when they are executed in that order. If the order is changed, they may have another commutativity condition. For example, the counter must be positive for the method pair decrease/increase. These methods always commute because executing the increase before the decrease will never break the positive invariant. However, on the other hand, for the method pair increase/decrease, these methods only commute if the counter is bigger than 0 because executing the decrease before the increase may break the positive invariant.

Then, we look for the **NOT**, **AND**, and **OR** tokens. If the **NOT** is found, we create the Not object (! *expr*), and the remainder is processed. For the other two, the object And or Or is created depending on the token that has been found (*expr*1 && *expr*2 && ...), and the remainder expression is subdivided.

The next step is two extract binary operations. Thus we extract the comparators (**EQ**,**BIG**,**LOW**,**BIGE**,**LOWE**) and the math operations (**SUM**,**LESS**,**MULT**,**DIV**). This allows us to define the binary operations in Java. For example, $(= \ x \ 1)$ becomes $x \ = \ 1$. Finally, we define the variables and the operands. We look for the **TRUE**, **FALSE** tokens, in the case that a boolean object may require those values, the **NUM**, to identify the numeric operands, and finally, the variables, which are the **PARAMETERS**, which stands for the method parameter with the corresponding index, the **IDENTIFIER**, that represents the fields, and finally the **OTHER**, which is a field from an object that is passed as a parameter.

After the parse is created, an object which contains the methods (method1 and method2) and the commutativity condition as a String in Java syntax. These objects are stored in a set, and now their conditions will be converted to Java BiFunctions.

4.2.2.2 Convert Java String to BiFunction

The Java BiFunction was adopted because it was necessary to define the inputs of both methods. Thus we needed two inputs and the output, which returns if the methods do commute. For the Inputs, we expect an Object[], which has at the index 0 the *this* object, and the remaining are the parameters. For the output, a boolean is returned.

To convert the String to a function, a ScriptEngine is used, which is the "Nashorn,"which is a javascript engine.

First, the engine is initialized if it has not been yet. Then to define that the function it is supposed to generate is a biFunction, it is necessary to declare a String with the javascript function, which takes two arguments "function(m1Args, m2Args)", and after it is concatenated with the condition resulting in a string of type, *function(m1Args, m2Args)* ***condition***. Then it is only necessary to call the engine and pass the constructor of the java Bifunction with the full path, which for the Java biFunction is *java.util.function.BiFunction*, and as the constructor argument, it is used the previously declared javascript function string.

After the engine runs, we extract the java BiFunction, and we fill our commutativity Map. As we have a method pair(two methods) and the corresponding biFunction, we have created a Map which contains another Map inside. Thus for the outside Map, we have defined the key as a String which is the method1, and has the value of the other Map. For the inside Map, we defined the key as a String representing the method2, and the value is the BiFunction. Thus when inserting the BiFunction in the correct position, we execute a get call using the method1 as the argument. If it returns null, we create the Map <String,BiFunction>, then we execute the put call, which takes as key the method2 and as value the BiFunction. This procedure is done for every line of the result file. Thus it is possible to have conditions for the pair method1 method2 but not for the pair method2 method1. It means the condition is the same.

5

Evaluation

This chapter presents the experimental work and discusses its corresponding outcomes. We begin by presenting our goals and methodology in Sections 5.1 and 5.2. Then, we proceed, to presenting our experiments and discussing its results in Sections 5.3 to 5.5.

5.1 Goals

Our evaluation has two main goals. The first is to show the correctness of the effects extraction and the produced outputs of each stage of the analysis.

The second goal is related to the performance of the developed solution. We seek to comprehend how the analysis and time of the analysis scales when the size of the source code increases and investigate the impact of using objects versus primitive types on this duration. Additionally, we aim to gain insights into the distribution of the analysis duration across its internal stages: side effect extraction and commutativity analysis. Lastly, we aim to quantify the time required to determine if two method calls commute, with the intention of assessing the viability of our approach for use in runtime systems for data replication management.

5.2 Methodology

Our validation of correctness relied on the analysis of intermediate files and the examination of commutativity conditions files. We assessed if they were well-formatted and if the side effects were well-defined.

For our time analysis, we conducted ten executions and computed the average across these ten runs. To assess time scalability, we introduced new classes with an increased number of methods compared to the original class. The increment in the number of methods followed a specific factor: the second class had twice the number of methods as the original, the third had triple the number, and so forth, up to a factor of six. Additionally, we measured the time difference between two equivalent classes: one utilizing only primitive types and the other employing simple objects.

To compare the time consumption of the analysis' inner stages, namely *Effect Extraction* and *Commutativity Analysis*, we recorded intermediate time points during each execution. Consequently, we measured the time elapsed from the beginning until the end of *Effect Extraction*, and, at the conclusion of Commutativity Analysis, we measured the time once more.

The experiments regarding the time it takes to check if two method calls commute, were repeated one thousand times. We initiated a timer at the beginning of each verification process, and it stopped when the verification yielded a result.

All tests were conducted in a machine running Ubuntu 20.04.4 with Java 14 installed and the following specifications: Intel Core i7-7500U 2.7GHz dual-core processor with 8GB of memory. The version of ASM used was the version 9.

5.3 Effect Extraction Correctness and Stages Outputs

The primary focus concerning the correctness of the developed solution was to ensure that, for each method, the side effects were correctly inferred. To this end, we have applied the developed solution to five common use cases. For each use case, we present the extracted side effects for each method, the Servois input file (YAML) for each use case, and the output produced by Servois.

5.3.1 Account

The following example is a simple bank account (Listing 19) with a balance that cannot be negative. The class is composed of the constructor, which sets the initial account balance. There is a `deposit` method that takes as an argument an integer that represents the value we are depositing in the account. Thus the effect of this method is a simple balance increment by the parameter value. Method `withdraw` takes an integer as argument, which represents the amount it is going to be withdrawn. As with the previous method, this method produces a simple effect: the balance decrement by the amount referred to in the parameter. The `transfer` method receives two parameters: *to*, which is the account we are transferring to, and the amount that is the value to be transferred. This method invokes other methods, `withdraw` applied to the `this` object and `deposit` applied to the `to` object. The effects are the decrease in the balance by the amount and the increment on the *to* object balance.Method `accrueInterest` takes the interest rate as a parameter and increments the balance by the existing balance multiplied by the rate. Finally, we have to get methods that return the balance and thus do not produce effects. One is a get that does not require consistency (`getBalance`) and the other requires consistency (`getConsistent_Balance`). The side effects extraction generated YAML file and Servois results are respectively shown in Appendices II.1 and II.6.

As it is possible to observe in Table 5.1, method **getBalance()** always commutes with

```
public class Account {
    @Repl @Invariant("balance >= 0") private int balance;

    @MethodCondition(expression = "param0 > 0")
    public void deposit(int amount) {
        if (amount > 0) this.balance += amount;
    }

    @MethodCondition(expression = "param0 > 0")
    public void withdraw(int amount) { this.balance -= amount; }

    @MethodCondition(expression = "param1 > 0")
    public void transfer(Account to, int amount) {
        withdraw(amount);
        to.deposit(amount);
    }

    @MethodCondition(expression = "param0 > 0")
    public void accrueInterest(int interestRate) {
        if (interestRate > 0) this.balance = this.balance + this.balance * interestRate;
    }

    public int getBalance() { return this.balance; }

    @ConsistentRead
    public int getConsistent_Balance() { return this.balance; }
}
```

Listing 19: Account Example

	$accrueInterest(y_1)$	$deposit(y_1)$	$getBalance()$	$getConsistent\_Balance()$	$transfer(y_1, y_2)$	$withdraw(y_1)$
$accrueInterest(x_1)$	$true$	$false$	$true$	$(not\ (>\ balance\ 0))$	$false$	$false$
$deposit(x_1)$	$false$	$true$	$true$	$false$	$(not\ (>\ y_2\ balance))$	$(not\ (>\ y_1\ balance))$
$getBalance()$	$true$	$true$	$true$	$true$	$true$	$true$
$getConsistent\_Balance()$	$(not\ (>\ balance\ 0))$	$false$	$true$	$true$	$false$	$false$
$transfer(x_1, x_2)$	$false$	$true$	$true$	$false$	...	$(not\ (>\ y_1\ balance))$
$withdraw(x_1)$	$false$	$true$	$true$	$false$	$(not\ (>\ y_2\ balance))$	...

Table 5.1: Account Commutativity Map

the other methods. Method **getConsistent_Balance()** always commutes with **getBalance()** and itself and commutes with **accrueInterest(int x1)** if $(not\ (>\ balance\ 0))$, although the previous condition defines a condition that only implies that the balance cannot be bigger than zero, as the balance is non-negative then, this means that they commute if the balance is zero.

Method **accrueInterest(int x1)** always commutes with itself. The **deposit(int x1)** always commutes with itself and always commutes left with the **withdraw(int y1)** and **transfer(Account y1, int y2)**. It commutes right with **withdraw(int y1)** if the withdraw amount is smaller than the balance, $(not\ (>\ y1\ balance))$ and it commutes right with the **transfer(Account y1,int y2)** if the transfer amount is smaller than the balance, $(not\ (>\ y2\ balance))$.

The method **withdraw(int x1)** commutes right with **transfer(Account y1,int y2)** if the transfer amount is smaller or equal than the balance,$(not\ (>\ y2\ balance))$, and it commutes left with **transfer(Account y1, int y2)** if the withdraw amount is not larger than the balance, $(not\ (>\ y1\ balance))$. It commutes with itself if the balance is equal to the

```java
public class Auction {
    private final UUID code;
    private final String description;
    private @Repl @Invariant("price > 0") float price;
    private @Repl boolean available = true;
    private @Repl int bidder;

    public Auction(UUID code, String description, float initialPrice) {
        this.code = code;
        this.description = description;
        this.price = initialPrice;
    }

    public float getPrice() { return this.price; }

    @ConsistentRead
    public float getCosistentPrice() { return price; }

    public int getBidder() { return bidder; }

    @ConsistentRead
    public int getConsistent_Bidder() { return bidder; }

    @MethodCondition(expression = "available = true && param1 > price")
    public boolean bid(int newBidder, float newPrice) {
        if (this.available && this.price < newPrice) {
            this.bidder = newBidder;
            this.price = newPrice;
            return true;
        }
        return false;
    }

    @MethodCondition(expression = "available = true")
    int sold() {
        if (this.available) {
            this.available = false;
            return this.bidder;
        }
    }
}
```

Listing 20: Auction Example

sum of the two withdraw amounts, $(= \ (- \ balance \ x1) \ y1)$ or if the previous condition is false and the amounts are smaller than the balance, $(and \ (not \ (= \ (- \ balance \ x1) \ y1)) \ (not \ (> \ x1 \ balance)) \ (not \ (> \ y1 \ balance)))$.

5.3.2 Auction

The next example is a simple Auction program that has five fields, three of them replicated (Listing 20). The replicated fields are the `bidder`, represented by an identifier of type int, the `price`, which represents the current price (the highest bid) of the item, and the `availability` status, of type boolean. The side effects extraction generated YAML file and Servois results are respectively shown in Appendix II.2 and Appendix II.7.

The first method is `bid`, which has two parameters, the bidder and the bid amount. If

	$bid(y_1, y_2)$	$getBidder()$	$getConsistent\_Bidder()$	$getPrice()$	$getConsistent\_Price()$	$sold()$
$bid(x_1, x_2)$	$false$	$true$	$(= x_1\ bidder)$	$true$	$false$	$false$
$getBidder()$	$true$	$true$	$true$	$true$	$true$	$true$
$getConsistent\_Bidder()$	$(= x_1\ bidder)$	$true$	$true$	$true$	$true$	$true$
$getPrice()$	$true$	$true$	$true$	$true$	$true$	$true$
$getConsistent\_Price()$	$false$	$true$	$true$	$true$	$true$	$true$
$sold()$	$false$	$true$	$true$	$true$	$true$	$false$

Table 5.2: Auction Commutativity Map

the auction is available and the parameter bid is bigger than the existing bid, this method updates the bidder and the bid. Method `sold` closes the auction, *i.e.*, sets the `available` flag to *false*, but this is only possible if the auction is open, i.e., the `available` field is true. Finally, as usual, we have the *getters*, which do not produce effects.

Again, as it is possible to observe on Table 5.2, the read operations without consistency always commute. Thus, the method **getPrice()** commutes with all the other methods, while the **getCosistentPrice** does not commute with the **bid(int y1, float y2)**, but commutes with all the others. **bid(int x1, float x2)** does not commute with sold nor itself.

5.3.3 Counter

The next example is a counter that cannot be negative. Thus, this class (Listing 21) has a field, the contents that is an int. The field has the invariant, which implies that the counter value has to be positive. It has the `increment` method, which increments the counter by one, and the `decrement`, which decrements the counter by one. There is a `reset` method that sets the counter to zero. Two read methods return the counter value. One may not return the consistent value, and the other forces the consistent return of the value counter. Additionally, we have a method multiply which takes the factor as a parameter, and this method was added to this class to test the multiplication analysis because, as mentioned before, in the initial Servois implementation, it was not supported. The side effects extraction generated YAML file and Servois results are respectively shown in Appendix II.3 and Appendix II.8.

As it is possible to observe in Table 5.3, the **read()** method (does not require consistency) commutes with all other methods as it was expected, once it did not require consistency and did not change the counter field. The **consistent_read()** method (requires consistency) does not commute with the **increment()** and **decrement()** methods. It always commutes with itself and with the **read()**, as was also expected because read/read operations always commute (Section 2.1.3). The **consistent_read()** commutes with **mult(int var)** if $(contents = contents * var)$, this means that the multiplication does not change the counter value($contents$). In the end, the **consistent_read()** commutes with the **reset()** if the counter is smaller than one $(1 > contents)$, as the contents has the invariant which requires that counter is non-negative, then this means they commute if the counter is zero.

```java
public class Counter {
    @Repl @Invariant("contents >= 0") int contents;

    public void mult(int param0){
        if (param0 >= 1) contents = contents*param0;
        else contents = 0;
    }

    public boolean increment(){
        contents++;
        return true;
    }

    public boolean decrement(){
        if (contents > 0) {
            contents--;
            return true;
        }
        return false;
    }

    public void reset(){ contents = 0; }

    public int read(){return contents;}

    @ConsistentRead()
    public int consistent_read(){ return contents; }
}
```

Listing 21: Counter Example

	increment()	*decrement()*	*reset()*	*read()*	*consistent_read()*	*mult*(y_1)
increment()	*true*	(*not* (> 1 *contents*))	*false*	*true*	*false*	(= y_1 1)
decrement()	*true*	*true*	*false*	*true*	*false*	(= y_1 1)
reset()	*false*	*false*	*true*	*true*	(> 1 *contents*)	(*not* (> 0 (∗ *contents* y_1)))
read()	*true*	*true*	*true*	*true*	*true*	*true*
consistent_read()	*false*	*false*	(> 1 *contents*)	*true*	*true*	(= *contents* (∗ *contents* y_1))
mult(x_1)	(= 1 x_1)	(*and*(> (∗ *contents* x_1) 0) (*not* (> x_1 1)))	*true*	*true*	(= *contents* (∗ *contents* x_1))	...

Table 5.3: Counter Commutativity Map

The **decrement()** always commutes with itself since the final state is always the same, and in case of breaking the invariant, the invariant is broken in both execution orders. It left commutes with **increment()** if the counter value is one or bigger. It commutes with **mult(int var)** if the multiplication is bigger than zero and the multiplication factor is not bigger than one. Although, it always commutes right with the **increment()** and commutes right with **mult(int var)** if the multiplication factor is one. The **reset()** always commutes with itself and commutes right with **mult(int var)** if the result of the multiplication is zero.

5.3.4 CourseWare

CourseWare is a simple example of evaluating the commutativity applied to Sets (Listing 22). We have three sets, the students and the courses. Initially each of the sets was a simple set os strings, however, after defining the datatypes(Section 4.2.1.4) we defined

```
public class Courseware {
    @Repl Set<Course> courseSet = new HashSet<>();
    @Repl Set<Student> studentSet = new HashSet<>();
    @Repl Set<Enroll> enrolSet = new HashSet<>();

    public void addStudent(String student) { studentSet.add(new Student(student)); }

    public void addCourse(String course) { courseSet.add(new Course(course)); }

    public void removeCourse(String course) { courseSet.remove(new Course(course)); }

    public void removeStudent(String student){ studentSet.remove(new Student(student)); }

    public Triple<Set<Student>,Set<Course>,Set<Enroll>> query() {
        return new Triple<>(studentSet,courseSet,enrolSet);
    }

    public void enrol(String student, String course) { enrolSet.add(new Enroll(student,course)); }
}
```

Listing 22: CourseWare Example

	$addCourse(y_1)$	$addStudent(y_1)$	$enroll(y_1,y_2)$	$query()$	$removeCourse(y_1)$	$removeStudent(y_1)$
$addCourse(x_1)$	*true*	*true*	*true*	*true*	$(not\ (=\ x_1\ y_1))$	*true*
$addStudent(x_1)$	*true*	*true*	*true*	*true*	*true*	$(not\ (=\ x_1\ y_1))$
$enroll(x_1,x_2)$	*true*	*true*	*true*	*true*	*true*	*true*
$query()$	*true*	*true*	*true*	*true*	*true*	*true*
$removeCourse(x_1)$	$(not\ (=\ x_1\ y_1))$	*true*	*true*	*true*	*true*	*true*
$removeStudent(x_1)$	*true*	$(not\ (=\ x_1\ y_1))$	*true*	*true*	*true*	*true*

Table 5.4: Courseware Commutativity Map

the courses and the students as sets of custom made types: `Student` and `Course`, respectively. Both types are classes that contain a single field of type String. In the Student class, this string describes the student's name and in the Course class, the string describes the course's name. The source code is available in Appendix I.2. The Enroll class is a String pair that combines a student with a course.

The `CourseWare` class comprises methods:

- `addCourse` and `removeCourse`, which affect the Courses set,
- `addStudent` and `removeStudent` that affect the Student Set,
- `enroll` that adds a (student, course) pair to the Enroll set, and, lastly,
- `query` returns all the sets.

The side effects extraction generated YAML file and Servois results are respectively shown in Appendix II.4 and Appendix II.9.

As shown in the Table 5.4, it is possible to observe that almost all the methods commute without any condition. The only methods pair that require condition are the addCourse with the removeCourse which requires that the parameters must be different, i.e. it will affect a different value in the set $(not\ (=\ x_1\ y_1))$. The same condition is applied to the pair addStudent and removeStudent.

```
public class MultiCounter {
    Counter a;
    Counter b;

    public void incA() { a.increment(); }

    public void incB() { b.increment(); }

    public void inc(){ a.increment(); b.increment(); }

    public void dec(){ a.decrement(); b.decrement(); }

    public void decA() { a.decrement(); }

    public void decB() { b.decrement(); }

    public void incAdecB() { a.increment(); b.decrement(); }

    public void decAincB(){ a.decrement(); b.increment(); }

    public void reset(){ a.reset(); b.reset(); }

    public void read(){
        a.read();
        b.read();
    }

    public void consistent_read(){
        a.consistent_read();
        b.consistent_read();
    }
}
```

Listing 23: MultiCounter Example

5.3.5 MultiCounter

The MultiCounter example is a simple class that has two fields that are not replicated. However, these fields are Counter objects. Thus, the content field in the Counter class is replicated. Thus as explained in Section 4.1.3, the effects are still extracted. There are a few methods that invoke the methods on the class' fields, and thus we evaluate the extraction of effects on primitive fields of a field object.

Analyzing the commutativity results (Figure 5.5 (obtained from the Servois output in Appendix II.10), it is possible to observe that all the increments with decrements commute. By looking at the analysis on Counter class, this should not occur once increments and decrements on commute in specific conditions. The reason for this behavior is that the invariants defined in the Counter class (Listing 21) for the field contents are not visible in the object's scope. Thus all these methods commute because there is no restriction (postcondition), and the final state is the same.

Apart from these methods, looking for the consistent_read, it only commutes with the read method and the reset in a specific condition, which is correct. Additionally, the read method commutes with all other methods.

	incA()	*incB()*	*inc()*	*decA()*	*decB()*	*dec()*	*incAdecB()*	*decAincB()*	*reset()*	*read()*	*consistent_read()*
incA()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
incB()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
inc()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
decA()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
decB()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
dec()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
incAdecB()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
decAincB()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*false*
reset()	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	...
read()	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
consistent_read()	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	...	*true*	*true*

Table 5.5: MultiCounter Commutativity Map

5.4 Compilation Times

There are methods that take longer to calculate commutativity because they are composed of more complex expressions, to evaluate the scalability, i.e., what is the impact in the compilation time when the number of the methods of a class increase, we have multiplied the class methods by a factor in each escalation in relation to the original class. The multiplication of methods was realized to avoid a biased approach by selecting a method that takes less or more time, once each method pairs has its time to be analyzed. Thus the scalability is analyzed based on a factor, which is **x1** initially and represents the original class. After these methods are duplicated (factor **x2**), the **x3** has the triple of the methods from the original class, and so on. This logic is applied till factor six. Additionally, to increase the accuracy of the measures, for each factor, there are ten executions, and in the end, it is calculated the average of the ten executions. An example of the scalability test script is presented in Appendix II.11.

By looking at the Table 5.6, although the number of methods of each class is very similar, the compilation times differ. The main reason for the difference between compilation times is the complexity of each method. The counter and the Account have multiplications and some calculations that take method parameters as operands, which increases the complexity of finding an expression to allow commutativity. Otherwise, the Auction only sets new values, which is simple, and the CourseWare adds an element to the Set which is also simple to evaluate. The MultiCounter uses some of the methods from the Counter class but this time invoked from a Counter object.

When the methods are duplicated, the compilation times increase on average 3.3 times. To be more exact, for the class Account, the compilation time increased by (2.9x),

	Auction		Account		Counter		Courseware		MultiCounter	
	#Methods	Time	#Methods	Time	#Methods	Time	#Methods	Time	#Methods	Time
x1	7	34	6	254	6	144	7	22	11	151
x2	13	106	12	740	12	480	13	84	22	572
x3	19	225	18	1603	18	1048	19	193	33	1330
x4	25	399	24	2847	24	1849	25	357	44	2435
x5	31	629	30	4586	30	2866	31	580	55	3915
x6	37	918	36	7488	36	4144	37	872	66	5817

Table 5.6: Compile time in seconds according to the number of methods and the class

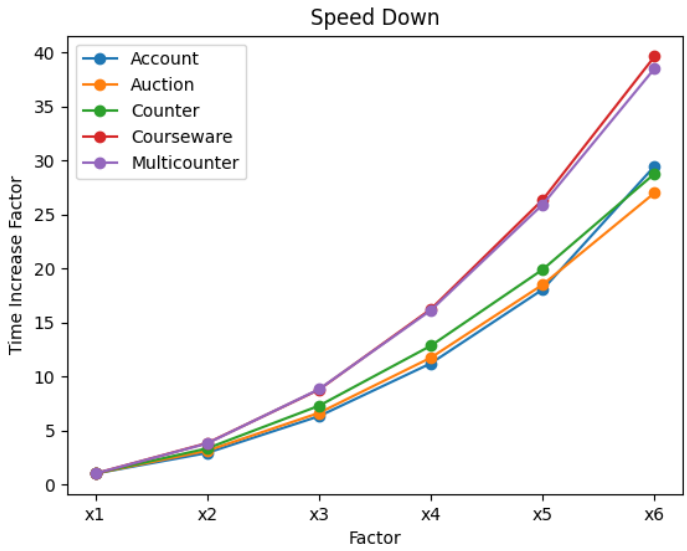

Figure 5.1: Scalability SpeedDown

for the Auction class (3.1x), for the Counter class (3.3x), for the CourseWare class (3.8x) and (3,8x) for the MultiCounter class.

When the number of methods is triplicated in relation to the original class, the compilation times on average increase (7 times). The class Account compilation time (x3 1603s) increases (6.3x) when compared to the original class.The compilation time for the Auction class (x3 225s) increases (6.6x).The Counter class compile time increase is (7.2x).For the CourseWare class, the compilation time increment is (8.7x) compared to the original class.Finally, for the MultiCounter class, the compilation time increment is (8.8x) compared to the original class.

When quadruplicated, the number of methods is compared to the original class and the compilation times increase in average (13.6) times. The compilation time for the classes Account, Auction, Counter, Courseware, and MultiCounter are 2847, 399s, 1849s, 357s, and 2435s, respectively. It shows an increase compared to the original class of (11.2x) for the class Account, (11.7x) for the class Auction, (12.8x) for the class Counter, (16.2x) for the class CourseWare, and (16.1x) for the MultiCounter class.

When the number of methods is five times more than the original class, the compilation time increases on average (21.7x). For the class Account, the increase is (18.0x); for the class Auction (18.5x); for the Counter class is (19.9x); for the Courseware class, the increase is (26.4x); and for the MultiCounter class, the increase is (25.9x).

Finally when the number of methods is six times the number of methods of the original class the compilation time increase on average (32.7x). The time increment for the Account class is (29.4x), it is (27x) for the Auction class, (28.8x) for the Counter class, (39.6x) for the Courseware class, and (38.5x) for the MultiCounter class.

By looking at Figure 5.1, which is a graphical representation of the time increase

factor calculated from Table 5.6 for each class, it is conclusive that the compilation time increases exponentially in relation to the number of equivalent methods.

The exponential growth in compilation time may be easily explained because when a method is added, this method has to be evaluated against all the other methods. Thus, if initially there were two methods, there would be only one evaluation, method1 against method2. If a method is inserted, there will be three evaluations, method1 with method2, method1 with method3, and method2 with method3. If a new method is added, there will be six evaluations. The number of evaluations is $(n-1)!$, which explains the exponential growth for compilation time.

Thus we can conclude that the compilation time is mainly affected by the complexity of the expressions that are deduced and evaluated for commutativity, but increasing the number of methods also increases the compilation time.

If the methods have equivalent expressions it is safe to conclude that the compilation times grow exponentially. Developers who adopt the REPL, should avoid using unnecessary methods and try to make the methods as straightforward as possible. Otherwise, compilation times may increase a lot.

5.4.1 Time Comparison Primitive vs Datatype

Additionally, we have evaluated the commutativity evaluation time comparison between two classes with the same methods, but one of them only has primitive fields (Listing 24) and the other an object which is converted to a datatype (Listing 25). We have removed the consistent reads once it is not supported for datatypes, and we have defined the price and the bidder as an object (Auction Bid Listing 26).

By looking for times in Table 5.7 and the Figure 5.2, it is possible to conclude that using datatypes for evaluating commutativity is more time-consuming than using primitive types. When the number of methods to analyze is reduced, the time difference is irrelevant, but when the number of methods is increased, the time difference increases. Thus, having primitive types is more beneficial when time performance is a concern.

	Primitive		Datatype	
	#Methods	Time	#Methods	Time
x1	5	15	5	17
x2	9	42	9	49
x3	13	86	13	101
x4	17	148	17	174

Table 5.7: Compile Time comparison in seconds according to the number of methods Auction class Primitive VS Datatype without consistent reads

```
public class AuctionT {
    private final UUID code;
    private final String description;
    private @Repl @Invariant("price > 0") float price;
    private @Repl boolean available = true;
    private @Repl int bidder;

    public AuctionT(UUID code, String description, float initialPrice) {
        this.code = code;
        this.description = description;
        this.price = initialPrice;
    }

    public float getPrice() { return price; }

    public int getBidder() { return bidder; }

    @MethodCondition(expression = "available = true && param1 > price")
    public boolean bid(int newBidder, float newPrice) {
        if (this.available && this.price < newPrice) {
            this.bidder = newBidder;
            this.price = newPrice;
            return true;
        }
        return false;
    }

    @MethodCondition(expression = "available = true")
    int sold(){
        if (available) {
            this.available = false;
            return this.bidder;
        }
    }
}
```

Listing 24: Primitive Types Auction Example without consistent reads

```
public class AuctionC {
    private final UUID code;
    private final String description;
    private @Repl AuctionBid bid;
    private @Repl boolean available = true;

    public AuctionC(UUID code, String description, float initialPrice) {
        this.code = code;
        this.description = description;
        this.available = true;
        bid = new AuctionBid(null,initialPrice);
    }

    public float getPrice() { return bid.getBid(); }

    public String getBidder() { return bid.getOwner(); }

    @MethodCondition(expression = "available = true")
    public boolean bid(String newBidder, float newPrice) {
        if (this.available && this.bid.getBid() < newPrice) {
            bid = new AuctionBid(newBidder,newPrice);
            return true;
        }
        return false;
    }

    @MethodCondition(expression = "available = true")
    AuctionBid sold() {
        if (available) {
            this.available = false;
            return this.bid;
        }
    }
}
```

Listing 25: Object Auction Example without consistent reads

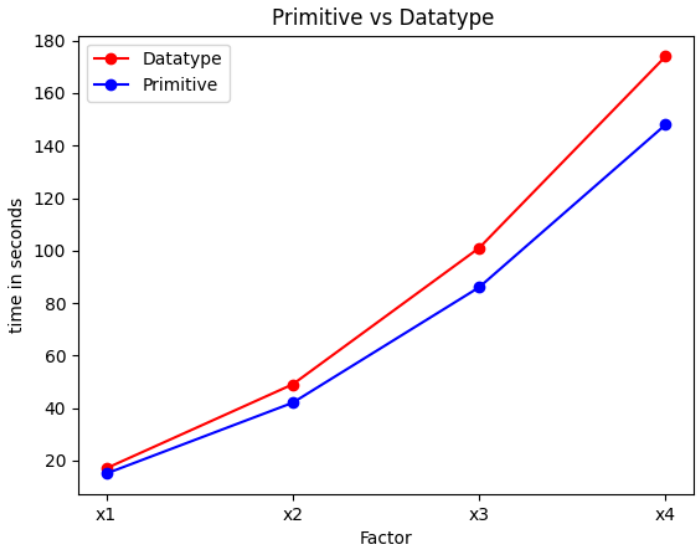

Figure 5.2: Time comparison scalability between a class using primitive fields and using object fields(primitive Auction 24, object Auction 25)

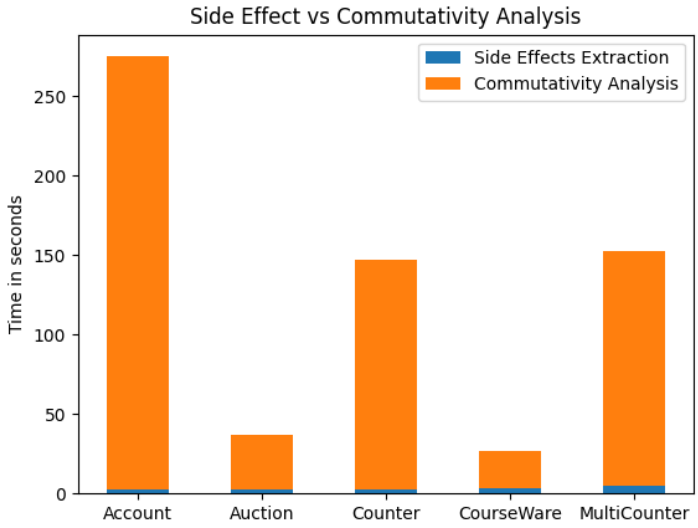

Figure 5.3: Comparison between time consumption on Effect Extractions vs Commutativity Analysis

5.4.2 Stages Comparison (Effects Extraction VS Commutativity Analysis)

We have extracted the the time taken for each stage, as it is possible to observe on Figure 5.3, the time taken by the Side Effects Extraction is almost non considerable when compared to the time taken by the Commutativity Analysis. Thus the main point of time optimization would be on the Commutativity Analysis.

```
public class AuctionBid implements Comparable<AuctionBid> {
    private float bid;
    private String owner;

    public AuctionBid( String owner,float bid){
        this.bid = bid;
        this.owner = owner;
    }

    public float getBid() { return bid; }

    public void setBid(float bid) { this.bid = bid; }

    public String getOwner() { return owner; }

    public void setOwner(String owner) { this.owner = owner; }

    @Override
    public int compareTo(AuctionBid that) {
        if (this.getBid() != that.getBid()) {
            return (this.getBid() < that.getBid() ? -1 : 1);
        }
        return 0;
    }
}
```

Listing 26: AuctionBid Class Example

5.5 Execution Time Verifications

The final metric to be evaluated is the verification time, which occurs after the condition map is built. It aims to measure the times taken to verify if two method calls fulfill the condition extracted from the conditions map. This evaluation is essential to understand if the effort taken during the compilation is worth the time during the execution, i.e., the verification time is significantly less than the average latency between replicated servers.

For this experiment, random methods were picked for each class being analyzed, and assertions were created to evaluate the correctness of the result of the execution of the lambda function. The time extraction test was executed 1000 times for each method pair analyzed to improve the accuracy and reduce the impact of outliers.

By analyzing the figure 5.4 we observe that the time to verify if two methods do commute is much lower when compared to the average latency between nodes in a distributed system. Although there are some outliers, we have a maximum latency of 0,03ms which is 10^2 times faster than the usual latency between systems. It is also possible to observe that 50% of the executions are between 0.005ms and 0.015ms and 25% below 0.05ms,

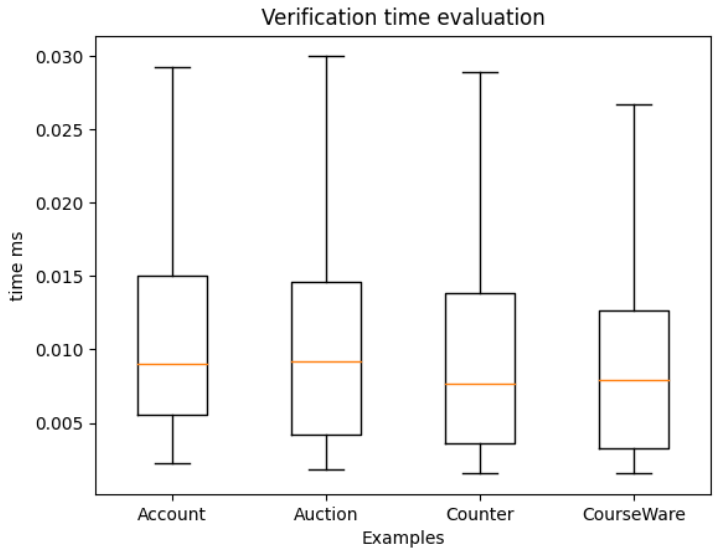

Figure 5.4: Condition verification times evaluation for the described Examples

representing that 75% of the executions are verified in less than 0.015ms. Additionally, despite the method or the class being verified, the latency barely changes; once for the presented examples, the times are very similar.

Considering this analysis, this experiment enforces that verifying if two methods commute during the runtime when the condition commutativity map is defined will almost have no effect on the latency times between the two systems. It may reduce the latency to a final client.

6

Conclusions

This work aimed to create a model capable of identifying commutativity between method calls. We had as starting point the previous work on the REPL project, which was able to evaluate if method calls commuted. The previous work only targeted primitive types and took advantage of the AST (Abstract Syntax Tree) to extract the method's side effects. In addition, our work, instead of using the AST as a base to side effect extraction, used the analysis of java bytecode and thus from each instruction to identify if it produced side effects. Our work also aimed to extract effects in simple objects and data structures and define expressions that make possible commuting method calls.

During the development of this work, one of the key concepts was to understand the Servois well. We started by creating some yaml files handmade to understand which expressions we needed to define and to create a mapping between the side effect and the SMT-Lib expression that would express the same goal. When working with datatypes, the Servois implementation did not introduce this term, so we had to research about the SMT-Lib datatypes and understand their behavior on Servois and the requirements to make it work. The implementation of Servois using Z3 was a significant change on the Servois core, although it supports creating more complex mathematical expressions, it could have behaved better when defining conditions for the Set (CourseWare example), as with everything in computer science, there are always trade-offs. The main reason for this behavior was because Servois was optimized for the CVC4, and Z3 produced some expression it did not recognize and thus did not support this condition.

The first approach to object effects extraction was to decompose the object till its primitive field and then for each primitive field, create a variable on Servois. This approach would increase the number of variables on Servois (states), and the variable names could be big once the though strategy was to describe the entire path from the starting object till the last primitive field. After researching about the SMT-Lib, the datatypes approach looked more correct. Although it was harder to implement, it reduced the expression size. After the development and considering that Servois was not designed to support datatypes, the first approach would be easier to implement once we would not need to declare datatypes, create constructors and use datatypes syntax.

```
public void deposit(int);
  Code:
     0: iload_1
     1: ifle          14
     4: aload_0
     5: dup
     6: getfield      #13                 // Field balance:I
     9: iload_1
    10: iadd
    11: putfield      #13                 // Field balance:I
    14: return
```

```
public void deposit(int amount) {
    if (amount > 0)
    this.balance += amount;
}
```

Figure 6.1: ByteCode instruction comparison to Java source code.

After some tests and discussions, we have pointed out some implementation limitations and some extensibility that could be performed.

6.1 Limitations and Future Work

Although we could extract side effects and evaluate commutativity on simple objects, it was not a general solution, i.e. it will only work for some types of objects. When working with objects, many corner cases increment the complexity of the analysis to extract the side effects. By changing the CVC4 to the Z3 solver, although it has increased capability to solve more complex math expressions, it has other limitations.

We have identified some limitations of the current solution and possible improvements for future work.

6.1.1 Conditional Effects Extraction

One of the keys for the future of this project is the identification of conditional effects, i.e., having different effects depending on some code conditions. The next step is identifying which operations are executed if a condition is fulfilled. By looking for Java bytecode when running the javap -c, it is possible to observe that when if clauses are inserted, there are conditional instructions, which, if they are not full-filled, jump to some instruction, Figure 6.1 shows, that if the if clause on java code is not fulfilled, it jumps to instruction 14: return.

One of the possibilities for future work is to analyze these conditional instructions and then group the following instructions into branches to determine if they fulfill the requirement or not.

Then when the effects are parsed to Servois, the notation **ite** should be used, which means if-then-else, and thus define the condition and the following effects for each branch. Another option is to use the Java API, which calls for the if-then-else clause for Z3 and could bring more benefits for the overall performance.

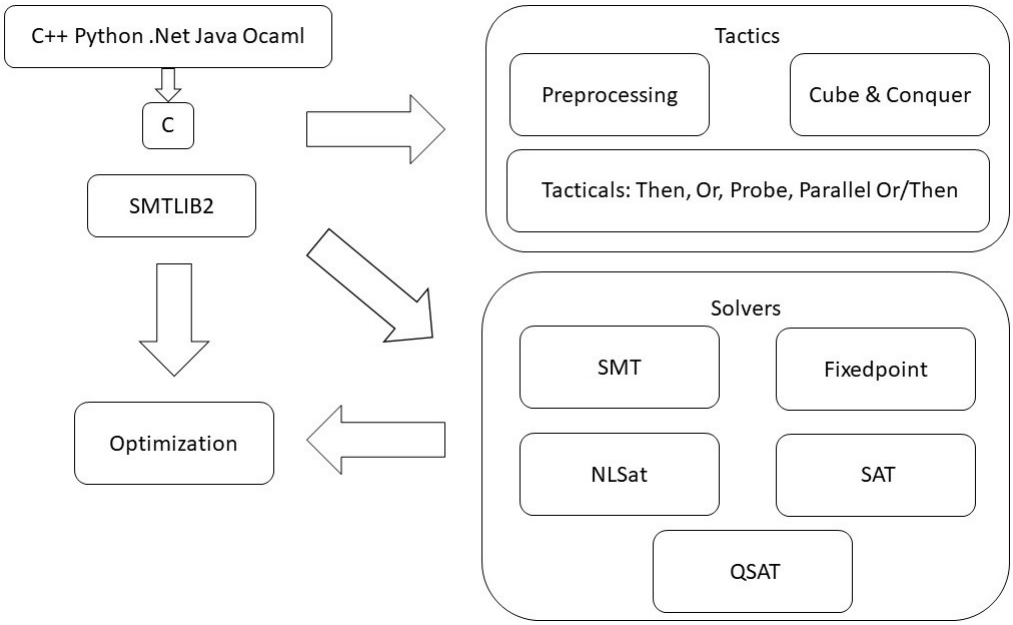

Figure 6.2: Z3 API system overview

6.1.2 Abstract Classes

Although our solution extracts side effects on abstract classes, we extract the abstract effect, i.e., the side effects are applied to the abstract variables. Thus when we have an object created from an abstract class, the object types and the side effects are mapped to the abstract objects instead of those used when instantiated the object. Thus, when objects are created from abstract classes in the future, it is necessary to create a mapping between the abstract variables and the instantiated variables and apply that mapping to the abstract side effects.

6.1.3 Servois Java Z3 API implementation

Looking at the actual development, there is a major step that could reduce the compilation time.

Instead of using the Servois tool, once it has its limitations and it is not optimized for the Z3 SMT engine, the development of a java program that has a similar approach to Servois but uses Z3 in its core, would benefit because as Z3 has an API for java, the effect could be translated automatically to the java Z3 Api object instead of creating a text file with all the specifications. This would remove the writes on the disk (yaml files). Thus this would save some time once writes on disk are expensive. Additionally, the results would be automatically returned through the API, and thus, Z3 would not require to write the results on disk, and thus one less read/write pair on disk would be performed and would reduce time consumption. This approach would also reduce the complexity of the developed solution once it eliminates the necessity to parse the different types of objects to smt-lib manually.

Another use case for implementing the Z3 API for Java is conditions applied to structures such as sets. In the CVC4 solver, it was possible to use the keyword **member**, which represented the contains, this expression would make it possible to validate if an object existed in the set. On the other hand, the Z3 Solver does not have that keyword. Instead, we have to create a selection for the object and an expression to evaluate if it exists. Once Servois was designed in its core with CVC4, the return of condition applied with the Z3 is not expected and thus fails.

Creating a custom Java program based on Servois techniques would allow the processing of more complex expressions, such as the previously presented ones, which Servois did not support.

6.1.4 Invariants and Method Conditions on Objects

One possible improvement is also the definition of invariants on objects. This is has two variants, defining invariants on an object's replicated fields or the invariant on the object. These steps would require adapting the JAVACC parser to support the objects. This would require understanding subfields and expressing a SMT-Lib expression condition in the object's subfield.

6.1.5 Consistent Read on Replicated Objects

Another possible improvement is to apply the consistent read on fields of replicated objects. When a method returns a field of a replicated object, it is necessary to define a strategy to extract that effect and define how it will be specified in SMT-Lib. As it is a field,initially, it is necessary to define a new state on Servois with the corresponding type. Then it is necessary to express the effect of the read on that particular object field. Thus it could be a complex expression such as the SMT-Lib datatype select. In the end Servois also would need some adjustments in its core to support the evaluation of this expression.

Bibliography

[1] H. N. S. Aldin et al. "Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications". In: *CoRR* abs/1902.03305 (2019). arXiv: 1902.03305. URL: http://arxiv.org/abs/1902.03305 (cit. on pp. 1, 2).

[2] F. Aleen and N. Clark. "Commutativity analysis for software parallelization: letting program transformations see the big picture". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*. Ed. by M. L. Soffa and M. J. Irwin. ACM, 2009, pp. 241–252. DOI: 10.1145/1508244.1508273. URL: https://doi.org/10.1145/1508244.1508273 (cit. on pp. 10, 18, 20, 25, 26).

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN: 1-55860-286-0 (cit. on p. 22).

[4] P. Bailis et al. "Coordination Avoidance in Database Systems". In: *Proc. VLDB Endow.* 8.3 (2014), pp. 185–196. DOI: 10.14778/2735508.2735509. URL: http://www.vldb.org/pvldb/vol8/p185-bailis.pdf (cit. on p. 4).

[5] V. Balegas et al. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. Ed. by L. Réveillère, T. Harris, and M. Herlihy. ACM, 2015, 6:1–6:16. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972 (cit. on pp. 3, 4).

[6] K. Bansal, E. Koskinen, and O. Tripp. "Synthesizing Precise and Useful Commutativity Conditions". In: *J. Autom. Reason.* 64.7 (2020), pp. 1333–1359. DOI: 10.1007/s10817-020-09573-w. URL: https://doi.org/10.1007/s10817-020-09573-w (cit. on pp. 7, 12, 13, 18, 23, 24, 26).

[7] C. W. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard Version 2.0*. 2010 (cit. on pp. 16, 17).

[8] E. A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. by G. Neiger. ACM, 2000, p. 7. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502 (cit. on p. 2).

[9] E. Bruneton, R. Lenglet, and T. Coupaye. "ASM: a code manipulation tool to implement adaptable systems". In: *Adaptable and extensible component systems* 30.19 (2002) (cit. on p. 41).

[10] J. Brutlag. *Speed Matters for Google Web Search*. URL: https://services.google.com/fh/files/blogs/google_delayexp.pdf (cit. on p. 1).

[11] R. Bruttomesso et al. "Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis". In: *Ann. Math. Artif. Intell.* 55.1-2 (2009), pp. 63–99. DOI: 10.1007/s10472-009-9152-7. URL: https://doi.org/10.1007/s10472-009-9152-7 (cit. on p. 17).

[12] M. Cole. "Algorithmic skeletons : a structured approach to the management of parallel computation". PhD thesis. University of Edinburgh, UK, 1988. URL: http://hdl.handle.net/1842/11997 (cit. on p. 10).

[13] *Consistency level choices*. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels. Accessed: 2023-09-23 (cit. on p. 2).

[14] G. DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by T. C. Bressoud and M. F. Kaashoek. ACM, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281 (cit. on p. 2).

[15] G. Dennis et al. "Automating commutativity analysis at the design level". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. Ed. by G. S. Avrunin and G. Rothermel. ACM, 2004, pp. 165–174. DOI: 10.1145/1007512.1007535. URL: https://doi.org/10.1145/1007512.1007535 (cit. on pp. 10, 11, 18, 20, 26).

[16] *Dynamic Analysis vs. Static Analysis*. URL: https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm (cit. on p. 15).

[17] T. Gehr, D. K. Dimitrov, and M. T. Vechev. "Learning Commutativity Specifications". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 307–323. DOI: 10.1007/978-3-319-21690-4_18. URL: https://doi.org/10.1007/978-3-319-21690-4%5C_18 (cit. on p. 24).

[18] S. Gilbert and N. A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on p. 2).

[19] A. Gotsman et al. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodık and R. Majumdar. ACM, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625. URL: https://doi.org/10.1145/2837614.2837625 (cit. on pp. 3, 4).

[20] A. Höfler. "SMT Solver Comparison". Graz, July 2014, Diploma Seminar (cit. on p. 17).

[21] F. Houshmand and M. Lesani. "Hamsaz: replication coordination analysis and synthesis". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 74:1–74:32. DOI: 10.1145/3290387. URL: https://doi.org/10.1145/3290387 (cit. on pp. 3, 4, 18, 22, 26).

[22] *Java BiFunction*. https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html. Accessed: 2023-09-23 (cit. on p. 35).

[23] T. J. K. E. von Koch et al. "Towards a compiler analysis for parallel algorithmic skeletons". In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. Ed. by C. Dubach and J. Xue. ACM, 2018, pp. 174–184. DOI: 10.1145/3178372.3179513. URL: https://doi.org/10.1145/3178372.3179513 (cit. on pp. 10, 13, 14).

[24] V. Kodaganallur. "Incorporating Language Processing into Java Applications: A JavaCC Tutorial". In: *IEEE Softw.* 21.4 (2004), pp. 70–77. DOI: 10.1109/MS.2004.16. URL: https://doi.org/10.1109/MS.2004.16 (cit. on pp. 33, 55, 65).

[25] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922 (cit. on p. 2).

[26] J. W. Lee, M. L. Soffa, and A. Zaks, eds. *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021. ISBN: 978-1-7281-8613-9. DOI: 10.1109/CGO51591.2021. URL: https://doi.org/10.1109/CGO51591.2021 (cit. on p. 18).

[27] S. S. Lunde. "Commutativity Analysis in ABS". MA thesis. UNIVERSITY OF OSLO, 2021 (cit. on p. 15).

[28] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24 (cit. on pp. xiv, 15).

[29] L. M. de Moura and N. S. Bjørner. "Model-based Theory Combination". In: *Proceedings of the 5th International Workshop on Satisfiability Modulo Theories, SMT@CAV 2007, Berlin, Germany, July 1-2, 2007*. Ed. by S. Krstic and A. Oliveras. Vol. 198. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2007, pp. 37–49. DOI: 10.1016/j.entcs.2008.04.079. URL: https://doi.org/10.1016/j.entcs.2008.04.079 (cit. on p. 17).

[30] *Nashorn project*. https://openjdk.org/projects/nashorn/. Accessed: 2023-09-23 (cit. on p. 35).

[31] E. M. Nystrom, H. Kim, and W. W. Hwu. "Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis". In: *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. Ed. by R. Giacobazzi. Vol. 3148. Lecture Notes in Computer Science. Springer, 2004, pp. 165–180. DOI: 10.1007/978-3-540-27864-1_14. URL: https://doi.org/10.1007/978-3-540-27864-1%5C_14 (cit. on p. 21).

[32] H. Paulino et al. "From atomic variables to data-centric concurrency control". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Ed. by S. Ossowski. ACM, 2016, pp. 1806–1811. DOI: 10.1145/2851613.2851734. URL: https://doi.org/10.1145/2851613.2851734 (cit. on p. 9).

[33] P. Prabhu et al. "Commutative set: a language extension for implicit parallel programming". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by M. W. Hall and D. A. Padua. ACM, 2011, pp. 1–11. DOI: 10.1145/1993498.1993500. URL: https://doi.org/10.1145/1993498.1993500 (cit. on pp. 18, 21, 26).

[34] E. Raman et al. "Parallel-stage decoupled software pipelining". In: *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*. Ed. by M. L. Soffa and E. Duesterwald. ACM, 2008, pp. 114–123. DOI: 10.1145/1356058.1356074. URL: https://doi.org/10.1145/1356058.1356074 (cit. on p. 22).

[35] M. C. Rinard and P. C. Diniz. "Semantic Foundations of Commutativity Analysis". In: *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*. Ed. by L. Bougé et al. Vol. 1123. Lecture Notes in Computer Science. Springer, 1996, pp. 414–423. DOI: 10.1007/3-540-61626-8_55. URL: https://doi.org/10.1007/3-540-61626-8%5C_55 (cit. on pp. 10, 13, 15, 18, 19, 26).

[36] D. B. Terry et al. "Consistency-based service level agreements for cloud storage". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by M. Kaminsky and M. Dahlin. ACM, 2013, pp. 309–324. DOI: 10.1145/2517349.2522731. URL: https://doi.org/10.1145/2517349.2522731 (cit. on pp. 3, 4).

[37] C. Vasiladiotis et al. "Loop Parallelization using Dynamic Commutativity Analysis". In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. Ed. by J. W. Lee, M. L. Soffa, and A. Zaks. IEEE, 2021, pp. 150–161. DOI: 10.1109/CGO51591.2021.9370319. URL: https://doi.org/10.1109/CGO51591.2021.9370319 (cit. on pp. 25, 26).

[38] W. E. Weihl. "Commutativity-Based Concurrency Control for Abstract Data Types". In: *IEEE Trans. Computers* 37.12 (1988), pp. 1488–1505. DOI: 10.1109/12.9728. URL: https://doi.org/10.1109/12.9728 (cit. on pp. 9, 12).

[39] P. Wu and A. D. Fekete. "An Empirical Study of Commutativity in Application Code". In: *7th International Database Engineering and Applications Symposium (IDEAS 2003), 16-18 July 2003, Hong Kong, China*. Ed. by B. C. Desai and W. Ng. IEEE Computer Society, 2003, pp. 358–369. DOI: 10.1109/IDEAS.2003.1214956. URL: https://doi.org/10.1109/IDEAS.2003.1214956 (cit. on pp. 10, 11, 18, 19, 26).

Annex 1 classes

I.1 Manual Java Set Implementation

```
public class JavaSet implements ManualStructureInterface {

    public static final class Add implements MethodEffect {
        public Add() {}

        @Override
        public EffectsList getEffect(CallEffect callEffect, MethodCall methodCall) {
            Resource list = methodCall.getTarget().resource;
            EffectsList ef = new EffectsList();
            Resource bool_true = new ConstResource<Boolean>(
                PrimitiveType.Boolean, Visibility.LOCAL,true);
            StoreOp storeOp = new StoreOp(
                list,callEffect.getParameters().get(0).resource,bool_true);
            ef.add(new Effect(list,storeOp));
            //set_size effect
            return ef ;
        }
    }

    public static final class Remove implements MethodEffect {
        public Remove() {}

        @Override
        public EffectsList getEffect(CallEffect callEffect, MethodCall methodCall) {
            Resource list = methodCall.getTarget().resource;
            EffectsList ef = new EffectsList();
            Resource bool_false = new ConstResource<Boolean>(
                PrimitiveType.Boolean, Visibility.LOCAL,false);
            StoreOp storeOp = new StoreOp(
                list,callEffect.getParameters().get(0).resource,bool_false);
            ef.add(new Effect(list,storeOp));
            return ef ;
        }
```

```
    }


    private static final String className = "java.util.set";

    private static final String servois_Constructor = "(Set %s)";

    private static final Map<String, MethodEffect> methodEffects = new HashMap<>();


    static {
        Add addMethodEffect = new Add();
        Remove removeMethodEffect = new Remove();
        methodEffects.put("add(java.lang.Object)", addMethodEffect);
        methodEffects.put("remove(java.lang.Object)",removeMethodEffect);
    }

    public String className() {
        return className;
    }

    public String servois_Constructor(){
        return servois_Constructor;
    }

    public MethodEffect methodEffects(String methodSignature) {
        return methodEffects.get(methodSignature);
    }
}
```

I.2 CourseWare Types

```
public class Course {

    private String name;

    public Course(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    @Override
    public int hashCode() {
        return name.hashCode();
    }
}

public class Student {


    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

II

Annex 2 Aditional Files

II.1 Servois Input Yaml Account Example

```yaml
name: "Account"
state:
- name: "balance"
  type: "Int"
- name: "OTHER.balance"
  type: "Int"
- name: "read_balance"
  type: "Int"
methods:
- name: "accrueInterest"
  args:
  - name: "param0"
    type: "Int"
  ensures: "(and (= OTHER.balance_new OTHER.balance)
            (= read_balance_new read_balance)
            (= balance_new (+ balance (* balance param0)))
            (= result true))"
  requires: "(and (> param0 0) (>= balance 0))"
  terms:
    Int:
    - "0"
    - "(+ balance (* balance $1))"
    - "balance"
    - "(* balance $1)"
    - "$1"
  return:
  - name: "result"
    type: "Bool"
- name: "getConsistent_Balance"
  args: []
  ensures: "(and (= OTHER.balance_new OTHER.balance)
```

```
            (= balance_new balance)
            (= read_balance_new balance) (= result true))"
  requires: "true"
  terms:
    Int:
    - "balance"
    - "read_balance"
  return:
  - name: "result"
    type: "Bool"
- name: "getBalance"
  args: []
  ensures: "(and (= OTHER.balance_new OTHER.balance)
            (= read_balance_new read_balance)
            (= balance_new balance)
            (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
- name: "withdraw"
  args:
  - name: "param0"
    type: "Int"
  ensures: "(and (= OTHER.balance_new OTHER.balance)
            (= read_balance_new read_balance)
            (= balance_new (- balance param0))
            (= result true))"
  requires: "(and (> param0 0) (>= balance 0))"
  terms:
    Int:
    - "0"
    - "balance"
    - "(- balance $1)"
    - "$1"
  return:
  - name: "result"
    type: "Bool"
- name: "deposit"
  args:
  - name: "param0"
    type: "Int"
  ensures: "(and (= OTHER.balance_new OTHER.balance)
            (= read_balance_new read_balance)
            (= balance_new (+ balance param0))
```

```
            (= result true))"
  requires: "(and (> param0 0) (>= balance 0))"
  terms:
    Int:
    - "0"
    - "(+ balance $1)"
    - "balance"
    - "$1"
  return:
  - name: "result"
    type: "Bool"
- name: "transfer"
  args:
  - name: "param0"
    type: "(Account Int)"
  - name: "param1"
    type: "Int"
  ensures: "(and (= read_balance_new read_balance)
            (= balance_new (- balance param1))
            (= OTHER.balance_new (+ OTHER.balance param1))
            (= result true))"
  requires: "(and (> param1 0) (>= balance 0))"
  terms:
    Int:
    - "0"
    - "balance"
    - "(+ OTHER.balance $2)"
    - "(- balance $2)"
    - "$2"
    - "OTHER.balance"
  return:
  - name: "result"
    type: "Bool"
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
```

```
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "String"
  - "String"
stateEquals: "(and (= balance_1 balance_2)(= read_balance_1 read_balance_2))"
preamble: |
  (declare-datatypes ( ( Account 1) ) ( ( par ( A ) ( ( account ( balance_c A ) ) ) ) ) )
```

II.2 Servois Input Yaml Auction Example

```
name: "Auction"
state:
- name: "bidder"
  type: "Int"
- name: "price"
  type: "Real"
- name: "available"
  type: "Bool"
methods:
- name: "getPrice"
  args: []
  ensures: "(and (= available_new available) (= price_new price)
            (= bidder_new bidder) (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
- name: "bid"
  args:
  - name: "param0"
    type: "Int"
  - name: "param1"
    type: "Real"
  ensures: "(and (= available_new available) (= bidder_new param0)
            (= price_new param1) (= result true))"
  requires: "(and (and (= available true) (> param1 price)) (> price 0))"
  terms:
```

```
    Real:
    - "0"
    - "price"
    - "$2"
    Int:
    - "bidder"
    - "$1"
  return:
  - name: "result"
    type: "Bool"
- name: "<init>"
  args:
  - name: "param0"
    type: "String"
  - name: "param1"
    type: "String"
  - name: "param2"
    type: "Real"
  ensures: "(and (= bidder_new bidder) (= available_new true)
            (= price_new param2) (= result true))"
  requires: " (> price 0)"
  terms:
    Bool:
    - "available"
    - "true"
    Real:
    - "0"
    - "price"
    - "$3"
  return:
  - name: "result"
    type: "Bool"
- name: "sold"
  args: []
  ensures: "(and (= price_new price) (= bidder_new bidder)
            (= available_new false) (= result true))"
  requires: " (= available true)"
  terms:
    Bool:
    - "available"
    - "true"
    - "false"
  return:
  - name: "result"
    type: "Bool"
- name: "getBidder"
```

```
  args: []
  ensures: "(and (= available_new available) (= price_new price)
            (= bidder_new bidder) (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "Bool"
  - "Bool"
- name: "="
  type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "String"
  - "String"
- name: "="
  type:
  - "Real"
  - "Real"
stateEquals: "(and (= bidder_1 bidder_2)
  (= price_1 price_2)(= available_1 available_2))"
preamble: ""
```

II.3 Servois Input Yaml Counter Example

```
name: "Counter"
state:
- name: "contents"
  type: "Int"
- name: "read_contents"
  type: "Int"
methods:
- name: "decrement"
  args: []
  ensures: "(and (= read_contents_new read_contents)
            (= contents_new (- contents 1)) (= result true))"
  requires: " (>= contents 0)"
  terms:
    Int:
    - "0"
    - "1"
    - "contents"
    - "(- contents 1)"
  return:
  - name: "result"
    type: "Bool"
- name: "consistent_read"
  args: []
  ensures: "(and (= contents_new contents) (= read_contents_new contents)
            (= result true))"
  requires: "true"
  terms:
    Int:
    - "contents"
    - "read_contents"
  return:
  - name: "result"
    type: "Bool"
- name: "increment"
  args: []
  ensures: "(and (= read_contents_new read_contents)
            (= contents_new (+ contents 1)) (= result true))"
  requires: " (>= contents 0)"
  terms:
    Int:
    - "0"
    - "1"
    - "contents"
    - "(+ contents 1)"
```

```
  return:
  - name: "result"
    type: "Bool"
- name: "read"
  args: []
  ensures: "(and (= read_contents_new read_contents)
            (= contents_new contents)  (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
- name: "reset"
  args: []
  ensures: "(and (= read_contents_new read_contents)
            (= contents_new 0) (= result true))"
  requires: " (>= contents 0)"
  terms:
    Int:
    - "0"
    - "contents"
  return:
  - name: "result"
    type: "Bool"
- name: "mult"
  args:
  - name: "param0"
    type: "Int"
  ensures: "(and (= read_contents_new read_contents)
            (= contents_new (* contents param0)) (= result true))"
  requires: " (>= contents 0)"
  terms:
    Int:
    - "0"
    - "contents"
    - "(* contents $1)"
    - "$1"
  return:
  - name: "result"
    type: "Bool"
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
```

```
  type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "String"
  - "String"
stateEquals: "(and (= contents_1 contents_2)(= read_contents_1 read_contents_2))"
preamble: ""
```

II.4 Servois Input Yaml Courseware Example

```
name: "Courseware"
state:
- name: "enrolSet"
  type: "(Set (Enroll String String))"
- name: "studentSet"
  type: "(Set (Student String))"
- name: "courseSet"
  type: "(Set (Course String))"
methods:
- name: "removeCourse"
  args:
  - name: "param0"
    type: "String"
  ensures: "(and (= enrolSet_new enrolSet) (= studentSet_new studentSet)
            (= courseSet_new (store courseSet (course param0) false))
            (= result true))"
  requires: "true"
  terms:
    (Course String):
    - "(course $1)"
    Bool:
    - "false"
```

```
    String:
    - "$1"
    (Set (Course String)):
    - "courseSet"
    - "(store courseSet (course $1) false)"
  return:
  - name: "result"
    type: "Bool"
- name: "enrol"
  args:
  - name: "param0"
    type: "String"
  - name: "param1"
    type: "String"
  ensures: "(and (= courseSet_new courseSet) (= studentSet_new studentSet)
            (= enrolSet_new (store enrolSet (enroll param0 param1) true)) (= result true))"
  requires: "true"
  terms:
    (Set (Enroll String String)):
    - "(store enrolSet (enroll $1 $2) true)"
    - "enrolSet"
    Bool:
    - "true"
    String:
    - "$1"
    - "$2"
    (Enroll String String):
    - "(enroll $1 $2)"
  return:
  - name: "result"
    type: "Bool"
- name: "addStudent"
  args:
  - name: "param0"
    type: "String"
  ensures: "(and (= courseSet_new courseSet) (= enrolSet_new enrolSet)
            (= studentSet_new (store studentSet (student param0) true))
            (= result true))"
  requires: "true"
  terms:
    (Set (Student String)):
    - "studentSet"
    - "(store studentSet (student $1) true)"
    Bool:
    - "true"
    String:
```

```
    - "$1"
    (Student String):
    - "(student $1)"
  return:
  - name: "result"
    type: "Bool"
- name: "query"
  args: []
  ensures: "(and (= courseSet_new courseSet) (= enrolSet_new enrolSet)
            (= studentSet_new studentSet)  (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
- name: "addCourse"
  args:
  - name: "param0"
    type: "String"
  ensures: "(and (= enrolSet_new enrolSet) (= studentSet_new studentSet)
            (= courseSet_new (store courseSet (course param0) true))
            (= result true))"
  requires: "true"
  terms:
    (Course String):
    - "(course $1)"
    Bool:
    - "true"
    String:
    - "$1"
    (Set (Course String)):
    - "(store courseSet (course $1) true)"
    - "courseSet"
  return:
  - name: "result"
    type: "Bool"
- name: "removeStudent"
  args:
  - name: "param0"
    type: "String"
  ensures: "(and (= courseSet_new courseSet) (= enrolSet_new enrolSet)
            (= studentSet_new (store studentSet (student param0) false))
            (= result true))"
  requires: "true"
  terms:
    (Set (Student String)):
```

```
    - "studentSet"
    - "(store studentSet (student $1) false)"
    Bool:
    - "false"
    String:
    - "$1"
    (Student String):
    - "(student $1)"
  return:
  - name: "result"
    type: "Bool"
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "(Set (Student String))"
  - "(Set (Student String))"
- name: "="
  type:
  - "(Set (Course String))"
  - "(Set (Course String))"
- name: "="
  type:
  - "String"
  - "String"
- name: "="
  type:
```

```
  - "(Set (Enroll String String))"
  - "(Set (Enroll String String))"
stateEquals: "(and (= enrolSet_1 enrolSet_2)(= studentSet_1 studentSet_2)
              (= courseSet_1 courseSet_2))"
preamble: |
  (declare-datatypes ( ( Student 1) ) ( ( par ( A ) ( ( student ( name_c A ) ) ) ) ) )
  (declare-datatypes ( ( Enroll 2) ) ( ( par ( A B )
        ( ( enroll ( student_c A )( course_c B ) ) ) ) ) )
  (declare-datatypes ( ( Object 0) ) ( ( par ( ) ( ( object  ) ) ) ) )
  (declare-datatypes ( ( Course 1) ) ( ( par ( A ) ( ( course ( name_c A ) ) ) ) ) )
```

II.5 Servois Input Yaml Multicounter Example

```
name: "MultiCounter"
state:
- name: "b.read_contents"
  type: "Int"
- name: "a.contents"
  type: "Int"
- name: "a.read_contents"
  type: "Int"
- name: "b.contents"
  type: "Int"
methods:
- name: "decB"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents) (= a.contents_new a.contents)
            (= b.read_contents_new b.read_contents) (= b.contents_new
            (- b.contents 1)) (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "(- b.contents 1)"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "consistent_read"
  args: []
  ensures: "(and (= b.contents_new b.contents) (= a.contents_new a.contents)
            (= a.read_contents_new a.contents) (= b.read_contents_new b.contents)
            (= result true))"
  requires: "true"
  terms:
    Int:
```

```
    - "b.read_contents"
    - "a.contents"
    - "b.contents"
    - "a.read_contents"
  return:
  - name: "result"
    type: "Bool"
- name: "decA"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents) (= b.contents_new b.contents)
            (= b.read_contents_new b.read_contents) (= a.contents_new (- a.contents 1))
            (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "(- a.contents 1)"
    - "a.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "read"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents) (= b.contents_new b.contents)
            (= a.contents_new a.contents) (= b.read_contents_new b.read_contents)
            (= result true))"
  requires: "true"
  terms: {}
  return:
  - name: "result"
    type: "Bool"
- name: "dec"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
            (= b.read_contents_new b.read_contents)
            (= a.contents_new (- a.contents 1)) (= b.contents_new (- b.contents 1))
            (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "(- b.contents 1)"
    - "(- a.contents 1)"
    - "a.contents"
    - "b.contents"
  return:
```

```
  - name: "result"
    type: "Bool"
- name: "inc"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
           (= b.read_contents_new b.read_contents)
           (= a.contents_new (+ a.contents 1)) (= b.contents_new (+ b.contents 1))
           (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "a.contents"
    - "(+ b.contents 1)"
    - "(+ a.contents 1)"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "reset"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
           (= b.read_contents_new b.read_contents)
           (= a.contents_new 0) (= b.contents_new 0)
           (= result true))"
  requires: "true"
  terms:
    Int:
    - "0"
    - "a.contents"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "incAdecB"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
           (= b.read_contents_new b.read_contents)
           (= a.contents_new (+ a.contents 1)) (= b.contents_new (- b.contents 1))
           (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "(- b.contents 1)"
    - "a.contents"
```

```
    - "(+ a.contents 1)"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "decAincB"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
            (= b.read_contents_new b.read_contents)
            (= a.contents_new (- a.contents 1)) (= b.contents_new (+ b.contents 1))
            (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "(- a.contents 1)"
    - "a.contents"
    - "(+ b.contents 1)"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
- name: "incA"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
            (= b.contents_new b.contents)
            (= b.read_contents_new b.read_contents)
            (= a.contents_new (+ a.contents 1)) (= result true))"
  requires: "true"
  terms:
    Int:
    - "1"
    - "a.contents"
    - "(+ a.contents 1)"
  return:
  - name: "result"
    type: "Bool"
- name: "incB"
  args: []
  ensures: "(and (= a.read_contents_new a.read_contents)
            (= a.contents_new a.contents)
            (= b.read_contents_new b.read_contents)
            (= b.contents_new (+ b.contents 1)) (= result true))"
  requires: "true"
  terms:
    Int:
```

```
    - "1"
    - "(+ b.contents 1)"
    - "b.contents"
  return:
  - name: "result"
    type: "Bool"
predicates:
- name: ">"
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "Int"
  - "Int"
- name: "<"
  type:
  - "Int"
  - "Int"
- name: "<="
  type:
  - "Int"
  - "Int"
- name: ">="
  type:
  - "Int"
  - "Int"
- name: "="
  type:
  - "String"
  - "String"
stateEquals: "(and (= b.read_contents_1 b.read_contents_2)(= a.contents_1 a.contents_2)
              (= a.read_contents_1 a.read_contents_2)(= b.contents_1 b.contents_2))"
preamble: |
  (declare-datatypes ( ( Counter 1) ) ( ( par ( A ) ( ( counter ( contents_c A ) ) ) ) ) )
```

II.6 Servois Output results Account Example

```
accrueInterest accrueInterest true
accrueInterest deposit false
accrueInterest getBalance true
accrueInterest getConsistent_Balance (not (> balance 0))
accrueInterest transfer false
accrueInterest withdraw false
deposit deposit true
deposit getBalance true
```

```
deposit getConsistent_Balance false
transfer deposit true
deposit transfer (not (> y2 balance))
withdraw deposit true
deposit withdraw (not (> y1 balance))
getBalance getBalance true
getBalance getConsistent_Balance true
getBalance transfer true
getBalance withdraw true
getConsistent_Balance getConsistent_Balance true
getConsistent_Balance transfer false
getConsistent_Balance withdraw false
transfer transfer (or (and (> y2 (- balance x2)) (> (+ OTHER.balance y2) (- balance x2))
                  (not (> x2 balance)) (not (> y2 balance))) (and (> y2 (- balance x2))
                  (not (> (+ OTHER.balance y2) (- balance x2))) (not (> x2 balance))
                  (not (> y2 balance))) (not (> y2 (- balance x2))))
withdraw transfer (not (> y2 balance))
transfer withdraw (not (> y1 balance))
withdraw withdraw (or (= (- balance x1) y1) (and (not (= (- balance x1) y1))
                  (not (> x1 balance)) (not (> y1 balance))))
```

II.7 Servois Output results Auction Example

```
<init> <init> (= y3 x3)
<init> bid false
<init> getBidder true
<init> getPrice true
<init> sold false
bid bid false
bid getBidder true
bid getPrice true
bid sold false
getBidder getBidder true
getBidder getPrice true
getBidder sold true
getPrice getPrice true
getPrice sold true
sold sold false
```

II.8 Servois Output results Counter Example

```
consistent_read consistent_read true
consistent_read decrement false
consistent_read increment false
mult consistent_read (= contents (* contents x1))
```

```
consistent_read mult (= (* contents y1) contents)
consistent_read read true
consistent_read reset (> 1 contents)
decrement decrement true
increment decrement (not (> 1 contents))
decrement increment true
mult decrement (and (> (* contents x1) 0) (not (> x1 1)))
decrement mult (= y1 1)
decrement read true
decrement reset false
increment increment true
mult increment (= 1 x1)
increment mult (= y1 1)
increment read true
increment reset false
mult mult (or (and (> (+ contents 1) x1) (= (* contents y1) y1)
          (= (* contents x1) (* contents y1)))
          (and (> (+ contents 1) x1) (= (* contents y1) y1)
          (not (= (* contents x1) (* contents y1)))
          (not (> (- contents 1) (* contents x1)))
          (not (> 0 y1))) (and (> (+ contents 1) x1)
          (not (= (* contents y1) y1)) (= (* contents y1)
          (* contents x1))) (and (> (+ contents 1) x1)
          (not (= (* contents y1) y1)) (not (= (* contents y1)
          (* contents x1))) (not (> 0 x1))
          (not (> 0 y1))) (and (not (> (+ contents 1) x1))
          (not (> 0 (* contents y1)))))
mult read true
reset mult (not (> 0 (* contents y1)))
mult reset true
read read true
read reset true
reset reset true
```

II.9 Servois Output results Courseware Example

```
addCourse addCourse true
addCourse addStudent true
addCourse enrol true
addCourse query true
addCourse removeCourse (not (= y1 x1))
addCourse removeStudent true
addStudent addStudent true
addStudent enrol true
addStudent query true
addStudent removeCourse true
```

```
addStudent removeStudent (not (= y1 x1))
enrol enrol true
enrol query true
enrol removeCourse true
enrol removeStudent true
query query true
query removeCourse true
query removeStudent true
removeCourse removeCourse true
removeCourse removeStudent true
removeStudent removeStudent true
```

II.10 Servois Output results MultiCounter Example

```
consistent_read consistent_read true
consistent_read dec false
consistent_read decA false
consistent_read decAincB false
consistent_read decB false
consistent_read inc false
consistent_read incA false
consistent_read incAdecB false
consistent_read incB false
consistent_read read true
consistent_read reset (and (not (> b.contents 0)) (> 1 a.contents)
                     (not (> 0 b.contents)) (not (> b.contents a.contents)))
dec dec true
dec decA true
dec decAincB true
dec decB true
dec inc true
dec incA true
dec incAdecB true
dec incB true
dec read true
dec reset false
decA decA true
decA decAincB true
decA decB true
decA inc true
decA incA true
decA incAdecB true
decA incB true
decA read true
decA reset false
decAincB decAincB true
```

```
decAincB decB true
decAincB inc true
decAincB incA true
decAincB incAdecB true
decAincB incB true
decAincB read true
decAincB reset false
decB decB true
decB inc true
decB incA true
decB incAdecB true
decB incB true
decB read true
decB reset false
inc inc true
inc incA true
inc incAdecB true
inc incB true
inc read true
inc reset false
incA incA true
incA incAdecB true
incA incB true
incA read true
incA reset false
incAdecB incAdecB true
incAdecB incB true
incAdecB read true
incAdecB reset false
incB incB true
incB read true
incB reset false
read read true
read reset true
reset reset true
```

II.11 Scalability evaluation Script Counter Example

```
#!/bin/sh
cd ..
a=0

while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.CounterTest"
```

```
        -Dorg.gradle.java.home=/opt/jdk-14
done

a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.Counter1Test"
        -Dorg.gradle.java.home=/opt/jdk-14
done

a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.Counter2Test"
        -Dorg.gradle.java.home=/opt/jdk-14
done

a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.Counter3Test"
        -Dorg.gradle.java.home=/opt/jdk-14
done

a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.Counter4Test"
        -Dorg.gradle.java.home=/opt/jdk-14
done

a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
   ./gradlew clean :repl-compiler:test --tests "repla.repl.effects.counter.Counter5Test"
        -Dorg.gradle.java.home=/opt/jdk-14
done
```