Diogo João Cardoso

A Methodology for Opacity verification for Transactional Memory algorithms using Graph Transformation System

Proposta de Tese apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Advisor: Prof. Dr. Luciana Foss
Coadvisor: Prof. Dr. André Rauber Du Bois

Pelotas, 2021

ABSTRACT

With the constant research and development of Transactional Memory (TM) systems, various algorithms have been proposed, and their correctness is always an important aspect to take into account. When analyzing TM algorithms, one of the most commonly used correctness criterion is opacity, which infers that the algorithm only generates executions that observe consistent states of the shared memory. One of the main definitions of opacity also includes a graph characterization that represents the conflict between transactions of a history. This definition has been extended to subclasses of opacity, resulting in new correctness criteria. Other applications of graphs have been used to aid in correctness criteria verification, some examples explore memory store order, dependency on read operations, or a happens-before graph.

This thesis proposes a formal definition to demonstrate that a given TM algorithm only generates opaque histories using a Graph Transformation System. A methodology is introduced that consists of translating an algorithm into production rules that manipulate the state of a graph. With a set initial state of transactions, the result of this method consists of a state space that includes every possible order of execution of transactional operations that are automatically checked for opacity. Therefore all possible histories the algorithm can generate are checked for the safety property. The proposed approach has demonstrated the capability to deal with one of the challenges TM algorithms, their complexity. A case study of a complex algorithm that can perform partial rollbacks is also presented. The main contribution of this thesis is a generic formalization of transactional memory algorithms that supports some of its main characteristics, in terms of versioning and conflict detection, and also a formalization that can possibly be extended to new or existing graph characterization of other safety properties.

Keywords: Transactional Memory. Graph Grammar. Formalism. Algorithms.

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

To this day, the research and development of advances in multiprocessor programming still try to leverage processing power of multi-core systems. The synchronization of shared memory accesses such that safety and liveness properties are preserved is an inherent challenge associated with multiprocessor algorithms. Safety ensures that an algorithm is correct with respect to a defined correctness condition while liveness ensures that the program threads terminate according to a defined progress guarantee (PETERSON; DECHEV, 2017).

Transactional Memory (TM) provides a high level concurrency control abstraction. At language level, TM allows programmers to define certain blocks of code to run atomically (HERLIHY; MOSS, 1993), without having to define *how* to make them atomic. Also, at implementation level, TM assumes that all transactions are mutually independent, therefore it only retries an execution in the case of conflicts. There are benefits of using TM over lock based systems, such as, composability (HARRIS et al., 2005), scalability, robustness (WAMHOFF et al., 2010) and increase in productivity (PANKRATIUS; ADL-TABATABAI, 2011). There are several proposals of implementations of TM: exclusively Software (SHAVIT; TOUITOU, 1997), supported by Hardware (HERLIHY; MOSS, 1993), or even hybrid approaches (DAMRON et al., 2006; MATVEEV; SHAVIT, 2015).

TM allows developing programs and reasoning about their correctness as if each atomic block executes a *transaction*, atomically and without interleaving with other blocks, even though in reality the blocks can be executed concurrently. The TM runtime is responsible to ensure correct management of shared state, therefore, correctness of TM clients depends on a correct implementation of TM algorithms (KHYZHA et al., 2018). A definition of what correctness is for TM becomes necessary when defining a correct implementation of TM algorithms. Intuitively, a correct TM algorithm should guarantee that every execution of an arbitrary set of transactions is indistinguishable from a sequential run of the same set. Several correctness criteria were proposed in the literature (GUERRAOUI; KAPALKA, 2008; DOHERTY et al., 2009; IMBS; RAYNAL, 2012; DOHERTY et al., 2013; LESANI; PALSBERG, 2014) and they rely on the concept of transactional histories. Recent works on *formal definitions* for TM focuses on consistency conditions (SIEK; WOJCIECHOWSKI, 2014; DZIUMA; FATOUROU; KANELLOU, 2015; KHYZHA et al., 2018; BUSHKOV et al., 2018), fault-tolerance (HIRVE; PALMIERI; RAVINDRAN, 2017; MARIĆ, 2017), and scalability (PELUSO et al., 2015; CLEMENTS et al., 2017).

Of the several correctness criteria proposed for TM, opacity is very well defined and known. The definition of opacity by Guerraoui; Kapalka (2008) allows for some sub-classes: Conflict Opacity (CO-Opacity) (KUZNETSOV; PERI, 2017) and Multi-version Conflict Opacity (MVC-Opacity) (KUMAR; PERI, 2015; KUMAR; PERI; VIDYASANKAR, 2014), etc. Opacity and its sub-classes use a graph characterization composed of a conflict graph that represents how the transactions relate to each other by some defined notion of conflict. A history, sequence of transactional events that have access to a shared memory, is considered correct if the conflict graph of its transactions presents no cycles.

This thesis aims to propose a methodology to define a Graph Transformation System (GTS) that represents a TM algorithm and demonstrate that, considering the notion of conflict introduced by (GUERRAOUI; KAPALKA, 2008), the algorithm only generates "correct" histories. As an initial result and proof of concept, a GTS was constructed and demonstrated that from a single history execution it is possible to generate its conflict graph and evaluate the correctness of the history (CARDOSO; FOSS; BOIS, 2019). The main goal of this thesis is to expand that idea for an entire TM algorithm that generated such history. Meaning that for a set of transactions, the aim of the methodology is to show that every execution observes a correct state of the shared memory. By covering every execution means that it includes every possible sequence of combinations of operations from the set of transactions. This is achieved by using a tool called GROOVE (RENSINK; DE MOL; ZAMBON, 2021), that allows for a powerful logic to aid in the creation of productions (set of rules that transform the state of the graph) with the help of generic labels and quantifiers. Without the use of GROOVE such GTS would be represented by a much more extensive definition. The main contribution of this thesis proposal is a methodology to formalize complex TM algorithms and its safety property check. This approach is capable of dealing with different characteristics of TM algorithms, in terms of versioning and conflict detection, which can make it a powerful tool for their correctness verification and also a formalization that can possibly be extended to new or existing graph characterization of other safety properties.

1.1 Structure

The remainder of this text is organized as follows. Chapter 2 presents the background knowledge for transactional memory. Chapter 3 describes the foundations for graph transformations. Chapter 4 presents the expected contributions of this proposal. Chapter 5 discuss the methodology for obtaining and evaluating the results of this thesis. Chapter 6 shows the schedule for the last two years of the Ph.D. program. Chapter 7 shows the current status of development of this thesis proposal. Finally, Chapter 8 presents a case study of the current development of the thesis.

2 TRANSACTIONAL MEMORY

High level programming languages relieve programmers of the need to work directly with assembly, the same way automated garbage collectors remove the concern of dynamic memory management. With abstraction in mind, transactional memory can be seen as a step towards effortless concurrent programming.

Transactional Memory (TM) borrows the abstraction of atomic transaction from the data base literature and uses it as a first-class abstraction in the context of generic parallel programs. TM only requires of the programmer the identification of which blocks of code must be executed in a atomic way, but not how the atomicity must be achieved. TM has been shown as an efficient way of simplifying concurrent application development. Besides being a simple abstraction, TM also demonstrates an equal performance (or even better) than refined and complex lock mechanisms.

Transactional memory enables processes to communicate and synchronize by executing *transactions*. A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer. Any number of operations on *transactional objects* (*t-objects*) can be issued, and the transaction can either *commit* or *abort*. When a transaction T commits, all its operations appear as if they were executed instantaneously (atomically). However, when T aborts, all its operations are rolled back, and their effects are not visible to any other transactions (GUERRAOUI; KAPAŁKA, 2010).

A TM can be implemented as a shared object with operations that allow processes to control transactions. The transactions, as well as t-objects, are then “hidden” inside the TM. Conflict detection between concurrent transactions may be eager, if a conflict is detected the first time a transaction accesses a t-object, or lazy when the detection only occurs at commit time. When using eager conflict detection, a transaction must acquire ownership of the value to use it, hence preventing other transactions to access it, which is also called pessimistic concurrency control. With optimistic concurrency control, ownership acquisition and validation only occurs when committing.

The next sections present the fundamentals of transactional memory. Section 2.1 presents the definition of transactions and histories. Section 2.2 describes the basic properties of TM and Section 2.3 the main correctness criterion used in this proposal.

Finally, Section 2.4 presents a summary of some related work on TM correctness verification with some examples of other methods and tools.

2.1 Transactions and Histories

A process can only access t-object via operations of the TM. Transactions and t-objects are referred to via their identifiers from the infinite sets $Trans = \{T_1, T_2, \dots\}$ and $TObj = \{x_1, x_2, \dots\}$. For clarity of representation, lowercase symbols such as x and y denote some arbitrary t-object identifiers from set $TObj$. These t-objects are also considered as though they only allow *read* and *write* operations and are referred to as *variables*.

Let $p_1, \dots, p_n$ be n processes (or threads) that have access to a collection of shared objects via atomic transactions. To realize operations in these objects the TM algorithm provides implementations of read, write, commit and abort procedures. These procedures are called transactional operations. A history *H* of a transactional memory contains a sequence of transactional operations calls.

A transactional operation starts its execution when a live process emits an invocation, and the operation ends its execution when the process receives a response. The responses of each procedure of a transaction *T* are the following:

- a successful commit returns C_T, otherwise if it fails it returns A_T, meaning transaction *T* aborted;
- an abort operation always returns A_T;
- a successful read returns the value requested from the shared memory, or A_T;
- a successful write operation returns ok, or A_T.

An invocation and the response of a transactional operation can be seen and treated as separate instances, for the sake of simplicity the remainder of this text will treat the response as if it happened immediately after the invocation. This way the representation of a read operation $read_1(x, 1)$, of transaction T_1, contains both the invocation *read(x)* and the response, value 1. A write is represented as $write_1(x, 2) \rightarrow ok$.

Definition 1 (Well-formed History). *Let H be any history, T be any transaction in H and* H|T *be the set that contains only operations of T in H. A history H is well-formed if for every transaction T, the following conditions are valid:*

- *the first event of event of* H|T *is an invocation;*
- *every invocation in* H|T*, that is not the last operation, is immediately followed by a corresponding response;*

- *every response in* H|T*, that is not the last operation, is immediately followed by an invocation;*
- *no event in* H|T *happens after* C_T *or* A_T*;*
- *if T' is a transaction in H executed by the same process that executes T, then the last event of* H|T *precedes the first event of* H|T' *in H, or the last event of* H|T' *precedes the first of* H|T*.*

2.2 Properties of Transactional Memory

A given TM history is said to preserve the real time ordering of execution if any transaction T_i that commits and updates a variable x before T_j starts, this way T_j cannot observe the old state of x. Guerraoui; Kapalka (2008) provide a formal definition of opacity and provide a graph-based characterization of such property in a way that a history is opaque only if the graph structure built from it is acyclic.

Sequential histories. A well-formed history *H* is *sequential* if no two transactions in *H* are concurrent. The correctness of sequential histories is trivial to verify, given a precise semantics of the shared objects and their operations.

Complete histories. A well-formed history *H* is *complete* if *H* does not contain any live transaction. This means that is possible to transform it to a complete history *H'* by committing or aborting the live transactions. For every history *H*, all complete histories *H'* is contained in the set *Complete(H)*.

Legal histories and transactions. Let *S* be any sequential history, such that every transaction in *S*, except possibly the last one, is committed. A history *S* is said to be *legal* if it respects the sequential specifications of all the shared objects.

2.3 Correctness Criteria

For an application, all operations of a *committed* transaction appear as if they were executed instantaneously at some single point in time. All operations of an *aborted* transaction, however, appear as if they never took place. From a programmer's perspective, transactions are similar to critical sections protected by a global lock: a TM provides an illusion that all transactions are executed sequentially, one by one, and aborted transactions are entirely rolled back.

However, hardly any TM implementation runs transactions sequentially. Instead, a TM is supposed to make use of the parallelism provided by the underlying multiprocessor architecture, and so it should not limit the parallelism of transactions executed by different processes. A real TM history thus often contains sequences of interleaved events from many concurrent transactions. Some of those transactions might be aborted because aborting a transaction is sometimes a necessity for optimistic TM protocols.

Several safety conditions for TM were proposed in the literature, such as opacity (GUERRAOUI; KAPALKA, 2008), Virtual World Consistency (IMBS; RAYNAL, 2012), TMS1 and TMS2 (DOHERTY et al., 2009) and Markability (LESANI; PALSBERG, 2014). There are also Serializability and Strict-Serializability (PAPADIMITRIOU, 1979), Causal Consistency and Causal Serializability (RAYNAL; THIA-KIME; AHAMAD, 1997), and Snapshot Isolation (BUSHKOV et al., 2013). All these conditions define indistinguishably criteria and set correct histories generated by the execution of TM. The safety property (ALPERN; SCHNEIDER, 1985; LYNCH, 1996) for a concurrent implementation informally requires that nothing "bad" happens at any point in any execution. If it does happen, there is no way to fix it in the future, which implies that a safety property must be *prefix-closed*: every prefix of a safe execution must also be safe.

A correctness criterion is a set of histories prefix-closed, in other words, the prefixes of every history is also correct, satisfying the criterion. An implementation, however, satisfies a correctness criterion *P* if all of its histories also satisfy criterion *P*.

2.3.1 Opacity

There are two important characteristics of the safety property for TM implementations: (1) transactions that commit must result in a total order consistent with a sequential execution; (2) it is desired that even transactions that abort have access to a consistent state of the system (resulted from a sequential execution).

The opacity correctness criterion was firstly introduced by Guerraoui; Kapalka (2008) with the purpose of dealing with these two characteristics. In an informal way, opacity requires the existence of a total order for all transactions (that committed or aborted). This total order is equivalent to a sequential execution where only committed transactions make updates.

It is worth noting that the original opacity definition (GUERRAOUI; KAPALKA, 2008) is not considered a safety property, because it is not prefix-closed. This was later refined in (GUERRAOUI; KAPAŁKA, 2010) filtering non prefix-closed histories and making opacity in fact a safety property.

Definition 2 (Final-state Opacity (GUERRAOUI; KAPAŁKA, 2010))**.** *A finite TM history H is final-state opaque if there exists a sequential TM history S equivalent to any H' $\in$ Complete(H), such that*

- *S preserves $\prec_H^{RT}$, and*
- *every transaction $T_i \in$ S is legal in S.*

Definition 3 (Opacity (GUERRAOUI; KAPAŁKA, 2010))**.** *A TM history H is opaque if every finite prefix of H (including H itself if H is finite) is final-state opaque.*

By requiring that the sequential history S_p, of every prefix, be equivalent to a history $H'_p \in$ *Complete(H_p)*, means that opacity treats every transaction $T \notin$ *Complete(H_p)* as aborted. It is possible to call S_p as an opaque serialization of H_p. This definition of opacity is similar to du-opacity (ATTIYA et al., 2015).

2.3.2 Graph Characterization of Opacity

Guerraoui; Kapałka (2010) introduced a graph-based characterization of opacity with the purpose of being used to prove correctness of TM systems. From a history *H*, with only read and write operations, a graph is constructed representing the conflict dependencies between transactions in *H*. The history *H* with consistent reads and unique writes is proven opaque if, and only if, the graph is acyclic.

Let x be any variable in any history *H*, the transactions in *H* that write to x create a *version* of x. Because two transactions can write to x concurrently, determining the order between versions in *H* is not obvious. However, this order can be determined by the *implementation* history of a given TM algorithm. An implementation history is made of the same operations in a normal history, but it also includes additional internal operations to the variables, i.e, the history includes what happens between the invocation and response of an operation. When building a graph representing dependencies between transactions in *H*, it is assumed that the version order of x in *H* is known.

The version order of a variable x in *H* is a total order $\ll_x$ over the set of transactions in *H* that:

- are committed or commit-pending, and
- write to x,

such that T_0 is the least element according to $\ll_x$. A *version order function* in *H* is any function that maps every variable x to a version order of x in *H*.

Definition 4 (Graph characterization of Opacity)**.** *Let H be any TM history with unique writes and $V_\ll$ any version order function in H. Denote $V_\ll(x)$ by $\ll_x$. The directed, labelled graph OPG(H, $V_\ll$) is constructed in the following way:*

1. *For every transaction T_i in H (including T_0) there is a vertex T_i in graph OPG(H, $V_\ll$). Vertex T_i is labelled as follows: vis if T_i is committed in H or if some transaction performs a read operation on a variable written by T_i in H, and loc, otherwise.*
2. *For all vertices T_i and T_k in graph OPG(H, $V_\ll$), $i \neq k$, there is an edge from T_i to T_k (denoted $T_i \rightarrow T_k$) in any of the following cases:*

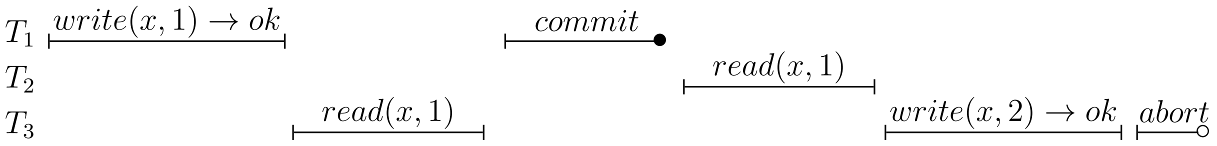
(a) If $T_i \prec_H T_k$ (i.e., T_i precedes T_k in H); then the edge is labelled rt (from "real-time") and denoted $T_i \xrightarrow{rt} T_k$;

(b) If T_k reads from T_i, meaning that T_i writes to the variable before T_k reads it; then the edge is labelled rf and denoted $T_i \xrightarrow{rf} T_k$;

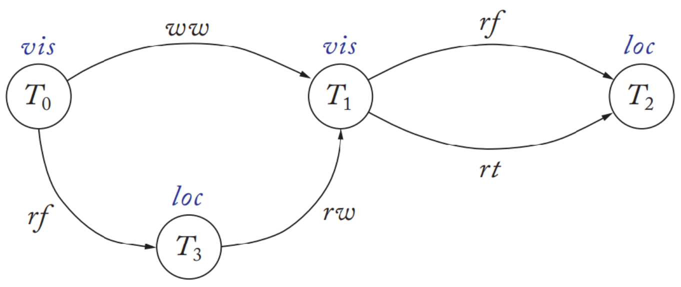
(c) If, for some variable x, $T_i \ll_x T_k$; then the edge is labelled ww (from "write before write") and denoted $T_i \xrightarrow{ww} T_k$;

(d) If vertex T_k is labelled vis, and there is a transaction T_m in H and a variable x, such that: (i) $T_m \ll_x T_k$, and (ii) T_i reads x from T_m; then the edge is labelled rw (from "read before write") and denoted $T_i \xrightarrow{rw} T_k$;

Figure 1 shows an example of a history *H* and its graph characterization *OPG(H,$V_\ll$)*. Note that transaction T_0, that writes the initial values in all variables of the TM, is not depicted in *H* because it is considered a default transaction to all histories. Transaction T_0 also finishes its execution before the first transactional operation of *H*.



(a) A history *H*.



(b) Graph Characterization of history *H*.

Figure 1 – A history (a) and its graph OPG(H,$V_\ll$) (b), where $V_\ll(x)$ = {(T_0, T_1)}. Source: (GUERRAOUI; KAPAŁKA, 2010).

Theorem 1 (Graph characterization of Opacity (GUERRAOUI; KAPAŁKA, 2010))**.** *Any history H with unique writes is final-state opaque if, and only if, exists a version order function $V_\ll$ in H such that the graph OPG(H,$V_\ll$) is acyclic.*

Proof. Proof can be found in (GUERRAOUI; KAPAŁKA, 2010). □

2.4 Related works on TM correctness

Several research work has been done on the correctness verification of transactional memory. Several approaches (EMMI; MAJUMDAR; MANEVICH, 2010; FLANA-

GAN; FREUND; YI, 2008; LITZ; DIAS; CHERITON, 2015) propose automatic techniques to verify correctness of transactional memory systems. Flanagan; Freund; Yi (2008) present the dynamic analysis tool Velodrome that performs atomicity verification that is both sound and complete. Velodrome analyzes operation dependencies within atomic blocks and infers the transactional happens-before relations of an observed execution trace. Serializability (PAPADIMITRIOU, 1979) of the execution trace is determined by verifying that the transactional happens-before graph is acyclic. Emmi; Majumdar; Manevich (2010) present an automatic verification method to check that transactional memories meet the correctness property strict-serializability (PAPADIMITRIOU, 1979). Their technique takes into consideration the number of threads and shared locations of the TM implementation and construct a family of simulation relations that demonstrate that the implementation refines the strict serializability specification. Litz; Dias; Cheriton (2015) present a tool that automatically corrects snapshot isolation (BUSHKOV et al., 2013) anomalies in transactional memory programs. The tool promotes dangerous read operations in the conflict detection phase of the snapshot isolation TM implementation and forces one of the affected transactions to abort. The authors reduce the problem of choosing the read operation to be promoted to a graph coverage problem for a dependency graph focusing on read operations. Since these techniques verify correctness based on the low-level read/write histories of the transactions, they are not directly applicable to transactional data structures that utilize high-level semantic conflict detection.

Formal logic has also been proposed to verify correctness of transactional memory systems as seen in (BLUNDELL; LEWIS; MARTIN, 2006; COHEN et al., 2007; MANOVIT et al., 2006). Blundell; Lewis; Martin (2006) demonstrate that a direct conversion of lock-based critical sections into transactions can cause deadlock even if the lock-based program is correct. Their observations highlights safety violations that may be introduced in transactional programs but does not provide a methodology for detecting the resulting faulty behavior. Cohen et al. (2007) present an abstract model for specifying transactional memory semantics, a proof rule for verifying that the transactional memory implementation satisfies the specification, and a technique for verifying serializability and strict serializability for a transactional sequence. Since conflicts considered in the abstract model are defined at the read/write level, the approach is limited to transactional memory systems that synchronize at low-level reads and writes. Manovit et al. (2006) present a framework of formal axioms for specifying legal operations of a transactional memory system. The dynamic sequence of program instructions called in the test are converted to a sequence of nodes in a graph, where an edge in the graph represents constraints on the memory order. The analysis algorithm constructs the graph based on the Total Store Order (TSO) memory model ordering requirements and checks for cycles to determine order violations. The graph construction is based on TSO ordering requirements, so the framework cannot be directly used to verify transactional correctness conditions that utilize high-level semantic conflict detection.

Peterson; Dechev (2017) present the first tool that can check the correctness of transactional data structures. The evaluation of correctness is based on an abstract data type, making the approach applicable to transactional data structures that use a high-level semantic conflict detection. The technique presented for representing a transactional correctness condition is a happens-before relation. The main advantage of this technique is that it enables a diverse assortment of correctness conditions to be checked automatically by generating and analyzing a transactional happens-before graph during model checking. Their work also present a strategy for checking the correctness of a transactional data structure when the designed correctness condition does not enforce a total order on a history. Serializability, strict serializability, and opacity require a total order on the history such that all threads observe the transactions in the same order. However, causal consistency requires only a partial order on a history, allowing threads to observe transactions in a different order.

3 GRAPH GRAMMARS

Graphs and graph transformations represent the core of most visual languages (BARDOHL R.AND MINAS; TAENTZER; SCHURR, 1999). In fact, graphs can be naturally used to provide a structured representation of the states of a system, which highlights their subcomponents and their logical or physical interconnections. In Graph Grammars and *Graph Transformation Systems* (GTS), the modification of graphs is specified via graph transformation rules, also known as graph productions (CORRADINI et al., 1997). Each rule consists of a pair of graphs, called *left-hand side* (LHS) and *right-hand side* (RHS), which schematically define how a graph may be transformed into a new graph. The events occurring in the system, which are responsible for the evolution from one state to another, are modelled as the application of these transformation rules. Such a representation is not only precise enough to allow the formal analysis of the system under scrutiny, but it is also amenable of an intuitive, visual representation, which can be easily understood also by a non-expert audience (BALDAN et al., 2008).

Applying a graph transformation rule to a graph can be seen as replacing a subgraph corresponding to the rule's LHS with a copy of its RHS. More precisely, elements that are specified in both LHS and RHS are preserved by the rule application, elements specified only in the LHS are deleted, and elements specified only in the RHS are created. When a graph transformation rule is applied to a graph, this graph is called *host graph*, not be confused with the LHS and RHS of the rule which are also graphs.

Graph transformation systems are a flexible formalism for the specification of complex systems, that may take into account aspects such as object-orientation, concurrency, mobility and distribution (EHRIG; ROZENBERG; KREOWSKI, 1999). GTSs are specially well-suited for the formal specification of applications in which states involves not only many types of elements, but also different types of relations between them. Also, applications in which behavior is essentially data-driven, that is, events are triggered by particular configurations of the state.

The possibility of applying a graph transformation rule to a host graph underlies the condition that a subgraph corresponding to the rule's LHS can be found. Furthermore, it is also possible that multiple matching subgraphs exist in a host graph. In such a case, multiple rule applications of the same rule can be performed. These rule applications are not necessarily independent. It might be the case that a choice has to be made at which match to transform the host graph, e.g., when different matches overlap and each their respective graph transformation modifies element contained in the other match. A set of graph transformation rules together with an initial graph spans a transition system. In this transition system, states represent the reachable state graphs from the initial state, and transitions between states represent the application of production rules, that change the state graph from one to another. It is important to realize that the nondeterminism indicated by multiple outgoing transitions of a state has two sources: multiple rules may be applicable to a graph and they may potentially be applied at multiple matches.

There exist various approaches to realize graph transformations. The two notable algebaric approaches are *double pushout* (DPO) (CORRADINI et al., 1997) and *single pushout* (SPO) (EHRIG et al., 1997). Both approaches are based on category theory and the categorical term of a pushout. In DPO, a transformation is formalized via two pushouts in the category of graphs and (total) graphs morphisms. One of the pushouts realizes the deletion of elements and the other one realizes their addition. In SPO, only a single pushout is used, which is a pushout in the category of graphs and partial graph morphisms.

The next sections present the fundamentals of graph transformations. Section 3.1 lays the algebraic foundation. It introduces the notions of graphs, graph morphisms and pushouts. Section 3.2 explains the workings of graph transformations in the SPO approach. Section 3.3 shows the visual representation used in this entire thesis proposal, alongside with negative conditions and typed graphs.

3.1 Graphs, Graph Morphisms and Pushouts

A graph is a structure that represents a set of objects along with relations between them. In this thesis proposal, only directed graphs are considered.

Definition 5 (Graph)**.** *A (directed) graph $G = (V_G, E_G, src_G, tgt_G)$ consists of a set of nodes V_G, a set of edges E_G, and source and target functions $src_G, tgt_G : E_G \to V_G$.*

Relations between graphs can be expressed through graph morphisms. A graph morphism is a mapping of nodes and edges of one graph to nodes and edges of another graph, respectively. Such that the source and target nodes of edges are preserved. Morphisms are used in graph transformation rules to define which nodes and edges are created, deleted, or preserved when the rule is applied to a graph.

Definition 6 ((Partial) Graph Morphism)**.** *A graph morphism $f : G \to H$ between two* *graphs is a pair of mappings f = (f_E, f_V) with $f_E : E_G \rightarrow E_H$ and $f_V : V_G \rightarrow V_H$ that commutes with the source and target functions, i.e., $f_V \circ src_G = src_H \circ f_E$ and $f_V \circ tgt_G = tgt_H \circ f_E$. A graph morphism is called injective if f_E and f_V are injective and called isomorphic if f_E and f_V are bijective.*

A subgraph S of G, written $S \subseteq G$ or $S \hookrightarrow G$, is a graph where $V_S \subseteq V_G$, $E_S \subseteq E_G$, $src_S = src_G|_{E_S}$ and $tgt_S = tgt_G|_{E_S}$. Such that $src_G|_{E_S}$ ($tgt_G|_{E_S}$) is a set of edges contained in the mapping src_G (tgt_G) but restricted to E_S. A ***(partial) graph morphism*** *g from G to H is a total graph morphism from some subgraph dom(g) of G to H. The subgraph dom(g) is called the restricted domain of g. The range of a graph morphism $g' : G \rightarrow H$, written ran(g'), is a subgraph S' of H where $V_{S'}$ is the image set of g'_V and $E_{S'}$ is the image set of g'_E.*

The category having labeled graphs as objects and graph morphisms as arrow is called **Graph**. The graphs over a fixed labeling alphabet and the partial morphisms among them form a category denoted by **Graph$^{\mathbf{P}}$**.

The application of graph transformation rules is based on the concept of "gluing" graphs together. Two different graphs sharing a common subgraph can be glued together by adding the uncommon nodes and edges of both graphs to the common subgraph. This is formalized by the categorical notion of a pushout.

Definition 7 (Pushout)**.** *Let $f : A \rightarrow B$ and $g : A \rightarrow C$ be two morphisms in a category* ***C****. A pushout (D, f', g') over f and g is defined by a pushout object D and morphisms $f' : C \rightarrow D$ and $g' : B \rightarrow D$ such that*

- *$g' \circ f = f' \circ g$ and* (commutativity)
- *for all objects X and morphisms $h : B \rightarrow X$ and $k : C \rightarrow X$ with $h \circ f = k \circ g$, there is a unique morphism $x : D \rightarrow X$ such that $x \circ g' = h$ and $x \circ f' = k$.* (universal property)

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "="] & B \arrow[d, "g'"] \arrow[ddr, dashed, "h"] & \\
C \arrow[r, "f'"] \arrow[drr, dashed, "k"'] & D \arrow[dr, dashed, "x"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & {} \\
{} & {} & X
\end{tikzcd}$$

In the diagram above, *A* is the common subgraph. The pushout object *D* is the result of gluing *B* and *C* via *A*, f, and g. The commutativity ensures that all elements of *B* and *C* that have a common preimage in *A* are glued together in *D*. The universal property ensures that

- elements of *B* and *C* that do not have a common preimage in *A* are not glued together in *D* and
- *D* does not contain elements that exist neither in *B* nor *C*.

If elements of *B* and *C* were glued together in *D*, then there would exist a graph *X* for which no morphism x : *D* $\rightarrow$ *X* satisfies $x \circ g' = h$ and $x \circ f' = k$, because x had to map the glued element simultaneously to different elements in *X* for $x \circ g' = h$ and $x \circ f' = k$ to hold. If *D* did contain elements that exist neither in *B* nor *C*, there would also exist such a graph, e.g., a subgraph of *D* that does not contain these elements.

3.2 Single Pushout

In the single pushout approach, graph transformation rules are defined by only one morphism, this morphism directly maps from the LHS to the RHS. To allow the deletion of elements, this morphism is partial instead of total. Intuitively, elements of the LHS that are outside of the morphism's restricted domain are deleted, and elements of the RHS that are outside of the morphism's range are created.

Definition 8 (Graph Transformation Rule (SPO))**.** *A* graph transformation rule $p =$ *(L, R,* r*) consists of two graphs L and R, called* left-hand side *and* right-hand side*, and an injective partial graph morphism* r : *L* $\rightarrow$ *R, called* rule morphism*.*

In an SPO graph transformation rule, the rule morphism specifies both addition and deletion. Therefore, the pushout construction for the SPO approach is more complicated than for the DPO approach. In addition to the concept of gluing, it has to realize deletion.

Deletion is realized in the SPO approach by "equalizing" two partial morphisms that are defined on the same domain of definition but on different restricted domains. This is done by removing all elements from their range that have different preimages under both morphisms. This concept is formalized by the categorical notion of a co-equalizer.

The SPO approach constructs a specific co-equalizer (HARTMANIS et al., 2006). Its construction assumes that, for each element that is contained in the restricted domains of both morphisms, both morphisms map to the same image. It is sufficient for the construction of a pushout in **Graph$^{\mathbf{P}}$** in Definition 10.

Definition 9 (Specific Co-Equalizer in **Graph$^{\mathbf{P}}$**)**.** *Let* a, b : *A* $\rightarrow$ *B be two (partial) morphisms such that* $\forall x \in$ *dom(*a*)* $\cap$ *dom(*b*)* : $a(x)$ = $b(x)$*. The* co-equalizer *of* a *and* b ***Graph$^{\mathbf{P}}$** is the tuple (C,* c*) where*

- *C* $\subseteq$ *B is the largest subgraph of* [b(A) $\cap$ a(A)] $\cup$ [($\overline{a(\mathrm{A})} \cap \overline{b(\mathrm{A})}$)] *and*
- c : *B* $\rightarrow$ *C, with dom(*c*) = C, is the identity morphism on C.*

To construct *C*, the co-equalizer filters out from *B* all elements for which there is a preimage under one of the morphisms a or b that is not defined under the other morphism. Therefore, only elements remain in *C* that have either the same preimage(s) under both morphisms or no preimages at all.

The construction of a pushout **Graph$^{\mathbf{P}}$** is realized via two pushouts in **Graph** and a co-equalizer (HARTMANIS et al., 2006). The total morphisms for the two pushouts in **Graph** are defined in dependence on the partial morphism of the pushout in **Graph$^{\mathbf{P}}$**. The co-equalizer is used to realize the deletion in the construction of a pushout in **Graph$^{\mathbf{P}}$**.

Definition 10 (Pushout in **Graph$^{\mathbf{P}}$**)**.** *Let b : A $\rightarrow$ B and c : A $\rightarrow$ C be two partial graph morphisms. The* pushout *over b and c in **Graph$^{\mathbf{P}}$** always exists and can be constructed in three steps:*

1. *Construct the pushout (C', A $\rightarrow$ C', C $\rightarrow$ C') of the total morphisms dom(c) $\rightarrow$ C and dom(c) $\rightarrow$ A in **Graph**.* (gluing 1)
2. *Construct the pushout (D, B $\rightarrow$ D, C' $\rightarrow$ D) of the total morphisms dom(b) $\rightarrow$ A $\rightarrow$ C' and dom(b) $\rightarrow$ B in **Graph**.* (gluing 2)
3. *Construct the co-equalizer (E, D $\rightarrow$ E) of the partial morphisms A $\rightarrow$ B $\rightarrow$ D and A $\rightarrow$ C $\rightarrow$ C' $\rightarrow$ D in **Graph$^{\mathbf{P}}$**.* (deletion)

*The pushout over b and c in **Graph$^{\mathbf{P}}$** is the tuple (E, C $\rightarrow$ C' $\rightarrow$ D $\rightarrow$ E, B $\rightarrow$ D $\rightarrow$ E).*

$$\begin{tikzcd}
dom(c) \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(PO)"] & A \arrow[d] & dom(b) \arrow[l, hook'] \arrow[r] \arrow[d, phantom, "(PO)"] & B \arrow[d] & \\
C \arrow[r, hook] & C' \arrow[rr] & & D \arrow[r] & E
\end{tikzcd}$$

Since the pushout in **Graph$^{\mathbf{P}}$** always exists, there is no counterpart to the gluing condition in SPO. Therefore, the application of an SPO graph transformation rule can be defined as a pushout in **Graph$^{\mathbf{P}}$**.

Definition 11 (Graph Transformation (SPO))**.** *Let p = (L, R, r) be a graph transformation rule and the total morphism m : L $\rightarrow$ G a match of its LHS L to a graph G. The* graph transformation *from G to H via p at m, written G $\overset{p,m}{\Longrightarrow}$ H, is given by the pushout (H, r^*, m^*) over r and m in **Graph$^{\mathbf{P}}$**.*

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "m^*"] \\
G \arrow[r, "r^*"] & H
\end{tikzcd}$$

The morphisms r^* and m^* are called derivation morphism *and* co-match *of G $\overset{p,m}{\Longrightarrow}$ H, respectively.*

The match m and the rule morphism r correspond to A $\rightarrow$ C and A $\rightarrow$ B of Definition 10, respectively. Since the match of an LHS to a host graph is always total, the first pushout in **Graph** does not do anything. The second pushout in **Graph** adds elements, similar to the second pushout during the application of a DPO rule. Due to the second pushout, A $\rightarrow$ B $\rightarrow$ D and A $\rightarrow$ C $\rightarrow$ C' $\rightarrow$ D commute, which allows to contruct their co-equalizer. At the end, the co-equalizer deletes all elements for which there is a perimage under m that is not defined under r. Regardless of whether or not such an element has another preimage under m that is defined under r, the element is deleted. The existence of another preimage is not relevant, see Definition 9. To end up in a valid graph, the co-equalizer deletes dangling edges as well.

3.3 Visual representation, Negative Application Conditions and Types

In this thesis, graph transformation is used on the basis of one particular tool that is capable of providing fast, hands-on experience named GROOVE (RENSINK; DE MOL; ZAMBON, 2021). Graphs in GROOVE consist of labelled nodes and edges. An edge is an arrow between two nodes. Node labels can be either node types or flags; the latter can be used to model boolean conditions, which is true for a node if the flag is present and false if not. GROOVE can work either in an untyped or typed mode. In untyped mode there are no constraints on the allowed combinations of node types, flags and edges. In this thesis proposal typed mode is used: all graphs and rules must be well-typed, meaning that they can be mapped into a special type graph. This is checked statically for the start graph and rules (GHAMARIAN et al., 2012).

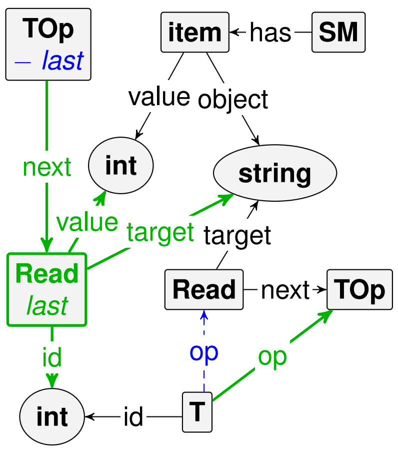


Figure 2 – Example of production rule using GROOVE.

Figure 2 shows a production rule for a transactional read operation used in this thesis proposal. Nodes and edges that are being created by the application of the rule (appear only in the RHS) or deleted (appear only in the LHS) are drawn in green and blue respectively. This rule specifies a read on the transactional memory by creating an element `Read` on the history, with the value read and target variable as edges.

To restrict the applicability of a rule, a *negative application condition* (NAC) can be used. A negative application condition forbids specific graph structures from being present in the host graph. Similar to the green or blue representation in Figure 2, a NAC is drawn in *red* instead.

Definition 12 (Negative Application Condition)**.** *Let $p = (L, R, r)$ be a graph transformation rule, G a graph and $m : L \to G$ a match. A* negative application condition (NAC) *is a tuple (N, n) where N is a graph and $n : L \to N$ an injective morphism. If there exists no morphisms $q : N \to G$ such that $q \circ n = m$, then m satisfies (N, n), written $m \models (N, n)$. Given a set of NACs $\mathcal{N}$, if $\forall (N, n) \in \mathcal{N} : m \models (N, n)$, then m satisfies $\mathcal{N}$, written $m \models \mathcal{N}$. For a graph transformation rule $p = (L, R, r)$ with a set of NACs $\mathcal{N}$, can also be written $p = (L, R, r, \mathcal{N})$.*

The above definition of NAC follows the SPO approach. A definition for DPO approach can be made analogously. The formal syntaxes and semantics of graph transformation rules can be built upon *typed graphs*.

Definition 13 (Typed Graph)**.** *Let TG be a distinguished graph, called* type graph*. A typed graph $G^T = (G, type)$ consists of a graph G = (V, E, src, tgt) and a total graph morphism $type : G \to TG$.*

The formal semantics of GROOVE is based on (typed) graph transformation systems and follows the SPO approach. While GROOVE will be used for the visual representation of graph transformation in this thesis proposal, most of the presented concepts also work in the DPO approach. Using the SPO approach rules are defined as an injective partial graph morphism. The use of injective morphisms implies that rules are not allowed to merge or to duplicate items.

4 CONTRIBUTIONS

The scope of the thesis proposal refers to the formalization of transactional memory algorithms, more specifically utilizing graph grammars and graph transformation to observe correctness of TM algorithms. When formally defining algorithms, usually, the intention is to check or prove that some properties hold for that algorithm, to guarantee correctness or safety, for example. A well known safety property for transactional memory is Opacity (described in Section 2.3.1), this property is specially suited for this thesis proposal because it already has a graph characterization well situated in the literature. This proposal aims to contribute in the research field of formalization of transactional memory algorithms. This proposal aims to introduce a novel approach to verification of opacity using a Graph Transformation System that utilizes graph characterization of opacity as safety check for the algorithm.

Specifically, the following contributions are expected:

- **Literature review:** a document discussing the sate of the art on transactional memory algorithms formalization, focusing on different approaches to correctness verification, and comparison between the most used ones. This result can be used to choose the most suitable correctness criterion for TM algorithms.

- **Generic formalization:** the definition of a methodology to formalize transactional memory algorithms that can be extended in regards to what algorithm characteristics are supported. Be either eager or lazy approaches to versioning and conflict detection, a global clock or the possibility of roll back are all possible features that can be represented in the graph transformation system.

- **Capability for new graph characterizations:** the approach of formalizing TM algorithms based on a graph characterization of a safety property opens the possibility of defining other correctness criteria with graph-based characterizations. Other approaches use either a happens-before graph or some sort of dependency graph to assist their proof of correctness. Having a full system based on graph transformations can be useful to adapt those methods and different criteria.

- **Proof of correctness:** concrete examples of correctness of TM algorithms using the proposed formalization via graph transformation. Demonstration of correctness of well established algorithms and newer algorithms with more complex features. In addition, counter-example algorithms that by definition do not satisfy opacity but can also be observed as not-opaque by the proposal.

The results of this thesis proposal are intended to contribute in the formalization of transactional memory algorithms, particularly in the verification of safety. This can be relevant for the definition of new algorithms that need a proof of correctness. Thus, the present research aims to provide a methodology to demonstrate correctness of TM algorithms with a well established formalization method.

5 METHODOLOGY

In order to propose a methodology of formalization of transactional memory algorithms via graph transformation systems, a research on correctness criteria and TM algorithms was first conducted. Analysing this research makes it possible to understand the relation between correctness criteria of TM algorithms and the methods used as proof of correctness. It was observed that a graph characterization of opacity, and its sub-classes, is a valid proof of correctness method so it stands to reason the idea of exploring a graph oriented formalism. This proposal aims to demonstrate the use of graph transformation systems to observe correctness of TM algorithms that are either classic and well established ones, or newer and more complex ones.

To support the proposed contributions, this thesis proposal was divided into several subgoals, as follows.

(1) **Study of graph characterization of Opacity:** This goal was performed as the central idea behind this thesis proposal, as the main motivator for the use of graph grammars and graph transformations. This results in the understanding the correctness dependant on conflict between transactions, and how they can affect the correctness verification through a graph characterization.

(2) **Graph Transformation System that supports Opacity:** The objective of this goal was to validate the idea of using graph transformation as a verification of transactional histories. A GTS that takes a single history as an input and observes its opacity was developed as a proof of concept resulting in a paper that was accepted for publication in the SBLP'19 proceedings.

(3) **Study of various Transactional Memory algorithms:** This goal was performed in the first Ph.D. year, where it was produced a document explaining the syntax and semantics of object synchronization languages, and various specifications of transactional memory algorithms. The production of this document was a requirement of the doctoral program.

(4) **Study of Transactional Memory correctness criteria:** This goal served as an adjunct of the initial idea of using graph transformations as formalization for TM.

It resulted in a study of the various correctness criteria for TM that showed that graph characterization and I/O automaton are the two main techniques to prove correctness. The produced text was presented during the qualification exam.

(5) **Graph Transformation System for a case study algorithm:** The objective of this goal was to create a first prototype of the thesis main contribution. A GTS was developed that observes opacity for a complex TM algorithm called CaPR+, this GTS generates all possible histories given a input of transactions. This resulted in a paper that was accepted for publication in SBLP'21 proceedings.

(6) **Definition of generic methodology of GTS for TM algorithms:** This goal aims to define a generic methodology of the proposal that approaches various characteristics of TM algorithms. The partial results are discussed in Chapter 7.

(7) **Graph Transformation System for TL2 algorithm:** The objective of this goal is to demonstrate that methodology of formalization proposed is able to observe opacity of classic algorithms like TL2.

(8) **Graph Transformation System for STM-Haskell algorithm:** This goal will be used as a counter-example of algorithm that does not satisfy opacity. This will demonstrate that the methodology proposed can observe opacity when it is satisfied and also observe the lack of opacity when it is not.

(9) **Graph Transformation System for Extra algorithm:** The objective of this goal will be to serve as an example of new algorithm that is uncertain if it is opaque or not.

The results of this thesis proposal is a methodology of formalization for TM algorithms that is flexible in terms of different characteristics of the algorithm and its possible complex features. This will be described as a set of generic rules that can adapt to different syntax and semantics derived from an algorithm.

After finishing the proposed goals and performing the steps of evaluation, the thesis will be written and defended.

6 SCHEDULE

The following timetable describes the main activities to be developed in order to finish the doctorate period.

Semester	Activities
2020/2	- Activity: Study of Transactional Memory correctness criteria and Qualification Exam
2021/1	- Activity: Graph Transformation System for a case study algorithm Paper: A Graph Transformation System formalism for correctness of Transactional Memory algorithms (Accepted in SBLP'21) - Defense of Thesis Proposal
2021/2	- Activity: Definition of generic methodology of GTS for TM algorithms - Activity: Graph Transformation System for TL2 algorithm - Paper: To be defined
2022/1	- Activity: Graph Transformation System for STM-Haskell algorithm - Activity: Graph Transformation System for Extra algorithm - Paper: To be defined
2022/2	- Thesis writing - Thesis defense

This schedule comprehends activities to be performed during the last two year of the Ph.D. program. The activities of semester 2021/1 were being completed while this document was written. The goals (1), (2), (3) and (4) presented in the Methodology chapter were already finished in the first two years of study, together with the obligatory courses to fulfill the total amount of credits for this program.

7 CURRENT STATUS

This chapter describes the methodology to formalize a transactional memory algorithm using a graph transformation system. The approach includes three main steps: first the translation of the logic of the algorithm to production rules, this step is made manually by analysing the procedures defined in the algorithm to create the state graphs desired; the second step is to generate all histories through a Labelled Transition System (LTS), this can be done with the help of GROOVE; and third, using the LTS and a graph characterization of a correctness criteria it is possible to use model checking to verify all histories resulted from the algorithm.

7.1 Translation to GTS

First, a representation of sequential operations is needed that will compose transactions and histories used in the entire approach. Figure 3 shows an example of two transactions with some conflicting operations, a code like this is the input for the system.

T_1	**T_2**
1: `begin`	1: `begin`
2: `  read(x)`	2: `  read(x)`
3: `  write(x,1)`	3: `  write(x,2)`
4: `tryCommit`	4: `tryCommit`

Figure 3 – Example of transaction code.

Figure 4 demonstrates how the code from Figure 3 is represented in a graph manner. This is the **initial state** as an input to the GTS. Each operation (`begin`, `read`, `write` and `tryCommit`) is represented by a node with relevant information to the operation itself, the main node `T` represents the identifier for the transaction with an unique id. Sequential operations are connected by an directed edge `next` that represents the order in which these operations must execute. The edge `op` represents the current operation to be executed, in a approach like this every transactional operation is considered to be atomic: the response of the operation happens immediately after the invocation. It was explored, in a previous work, a GTS formalism for histories where the invocations and responses are processed separately (CARDOSO; FOSS; BOIS, 2019). Because some of the correctness criteria for the TM algorithms use atomic operations, it was also chosen to be used in this approach, which in turns decreases the number of nodes in a transaction or history, making it more readable.

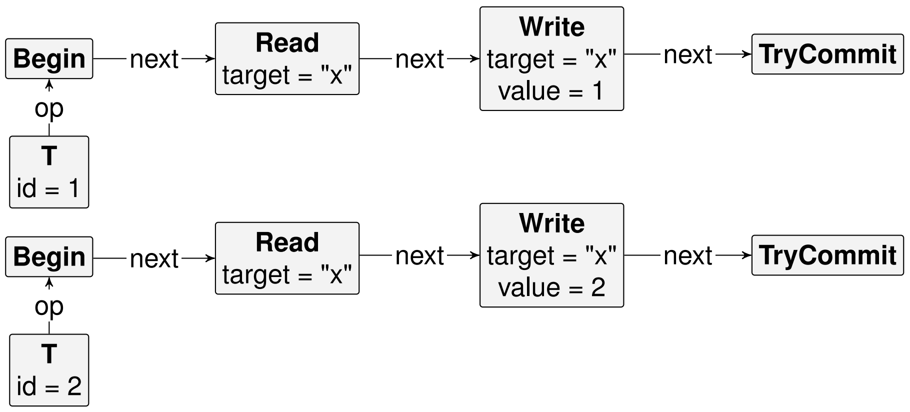


Figure 4 – Graph representation of transactions in Figure 3 as an input for the GTS.

7.1.1 Initial State and Type Graph

The initial state of the GTS includes the transactions (as seen in Figure 4) and some global objects like the shared memory, global clock, list of active transactions and so on. Which objects are treated globally or locally will depend on the algorithm itself. Figure 5 shows some examples of global objects that the algorithm logic will allow to be accessed at any moment. This restriction of access is enforced by the transformation rules.

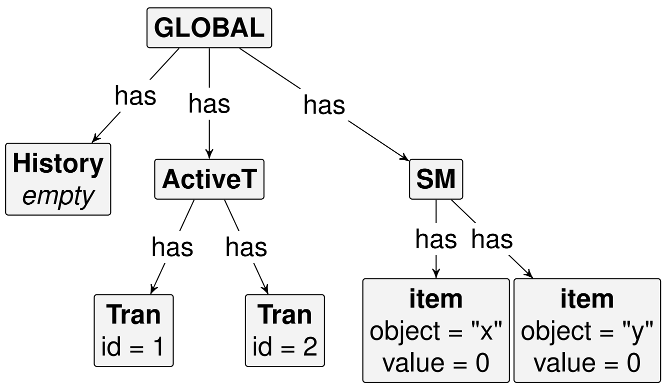


Figure 5 – Graph representation of global objects in the initial state of the GTS.

Another important aspect of the GTS formalism is the type graph. This special graph will determine what nodes and edges can exist in the system, this results in a controlled behavior by the production rules. Figure 6 shows an example of a type graph for the the global objects and initial state seen previously. In this example a feature of GROOVE called inheritance relation between nodes is used: the node `TOp` (Transactional Operation) is a sybtype of `Begin`, `Read`, `Write`, `TryCommit`, `Commit` and `Abort`. This is used mostly to simplify the relation that all of the transactional operation have with the nodes `History` and `T` (edges `has` and `op` respectively). The `TOp` node also has a recursive `next` edge that points to the next transactional operation, this is used for the sequential operations in the initial state and history.

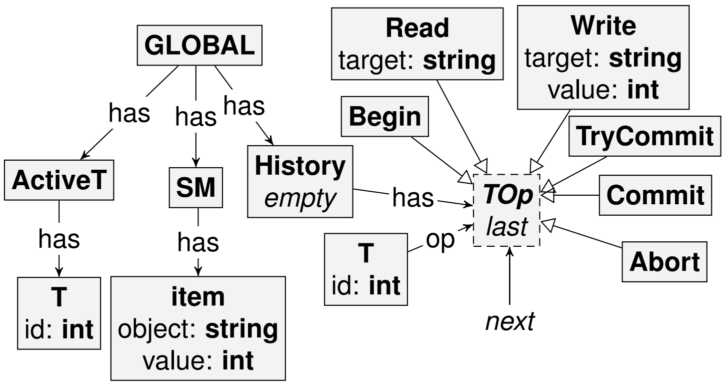


Figure 6 – Type Graph of the GTS.

Figure 7 show an example of a history after some transactional operations have been executed. It is very similar to the transactions seen in Figure 4.

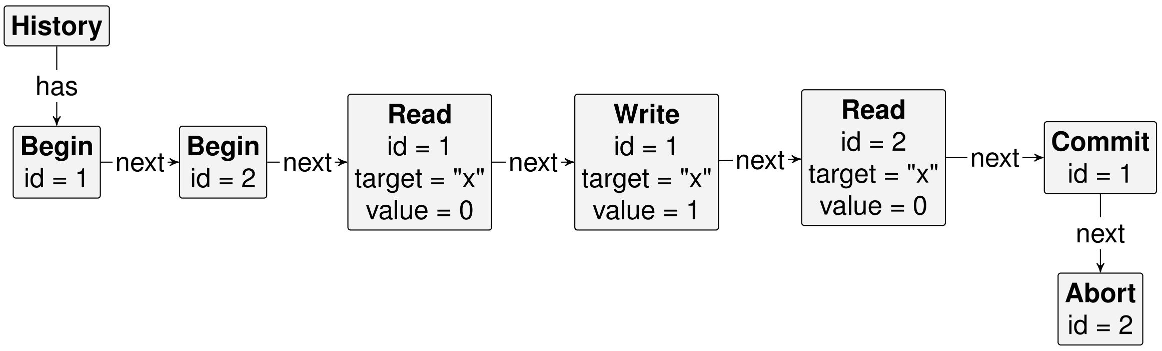


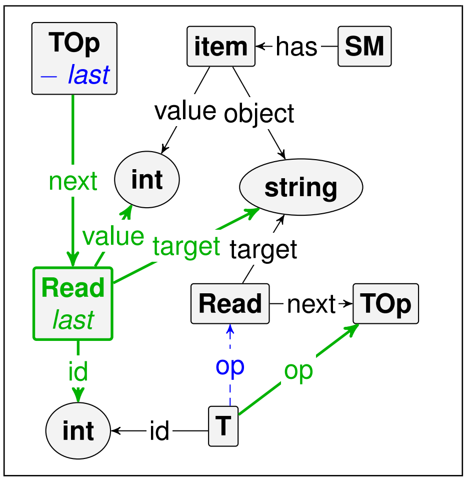
Figure 7 – Example of graph representation of a history for the GTS.

7.1.2 TM Procedures

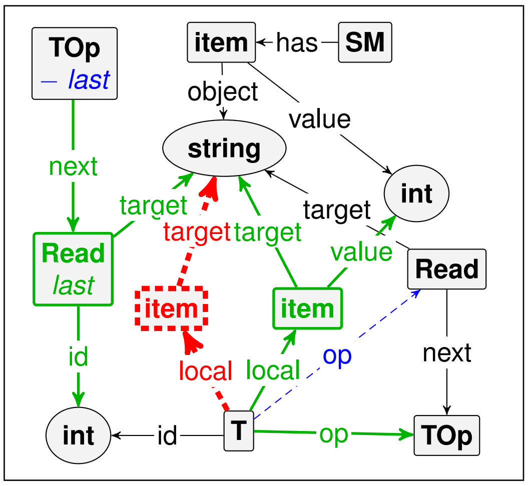
The next step in formalizing a TM algorithm with a GTS is dealing with the procedures of the algorithm. The main procedures used are: begin, read, write, commit and abort. Some algorithm might have extra procedures such as a rollback or verify operation, these will be discussed in later chapters.

The first operation described is a **read operation**. In Figure 8 it is showcased two different approaches to a read to the shared memory: an eager versioning in Figure

8(a) and a lazy one in Figure 8(b). These two examples manipulate the values of the shared memory (reading a specific variable) and create a new object in the last position of the history. Note that in the production rule for the lazy versioning read a NAC is used for the local item that will store the value read (and possibly written later). This ensures that this rule will only be executed if it is the first time this variable is being read from the shared memory. Any future read to the same variable will be done locally, with a modified production rule.



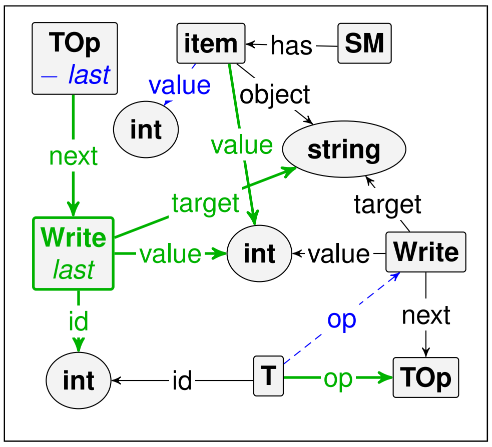
(a) Eager versioning



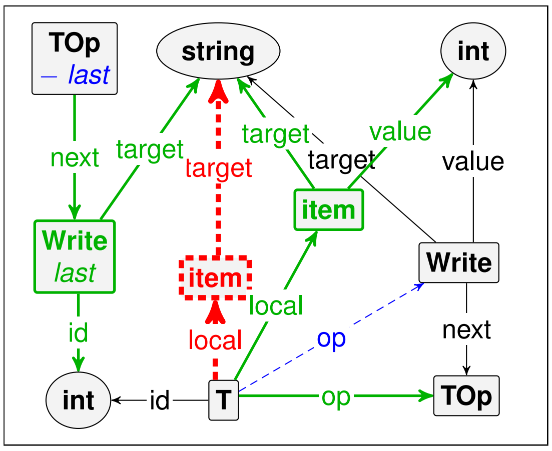
(b) Lazy versioning

Figure 8 – Example of production rules for a Read Operation.

The **write operations** are treated in a similar way. Figure 9 shows an eager and a lazy versioning approach to a production rule that executes a write. Similar to the rules seen above, the write operation also needs an extra production rule to execute the write locally if the local copy already exists.



(a) Eager versioning

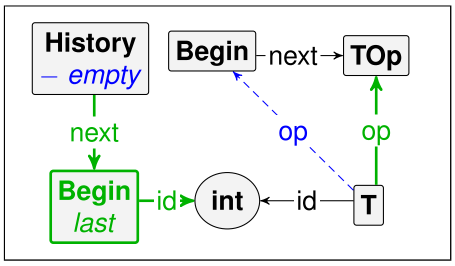


(b) Lazy versioning

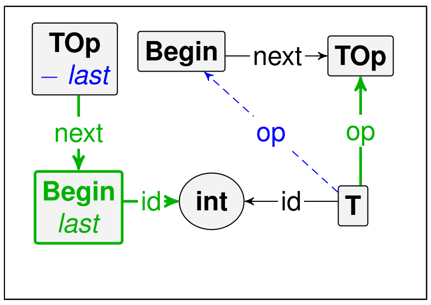
Figure 9 – Example of production rules for a Write Operation.

Some similarities can be seen between the production rules shown in Figures 8 and 9: an object is added at the end of the sequence representing the history (`TOp` loses the *last*-flag and points to a `Read/Write` with the flag instead); the `T` node that was pointing to the current operation with a *op*-edge (operation) now points to the next one in the sequence of *next*-edges; and lastly, the operations inside a transaction are not deleted for the possibility of being re-executed later.

The **begin operation**, that starts a transaction has two situations where it occurs: either it is the first operation of the entire system, therefore the history is empty; or the transaction is starting in the middle of the execution where the history is no longer empty. To accommodate for both situations two separate production rules are needed, as shown in Figure 10(a) and 10(b), and these two are the only rules required to deal with starting transactions.



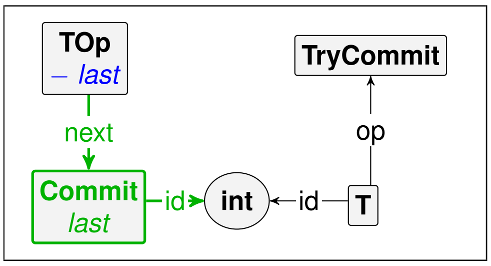
(a) Empty History



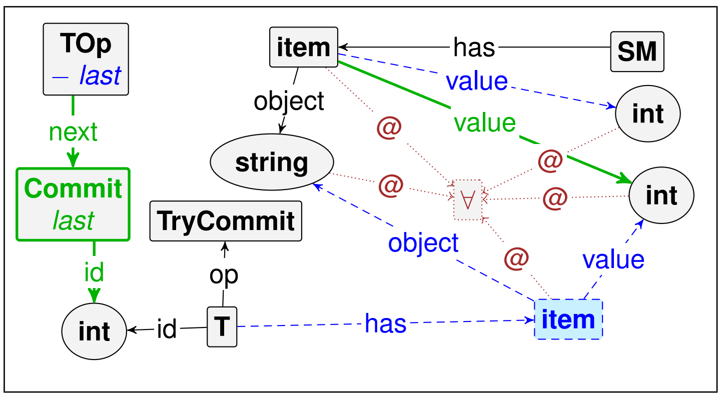
(b) Non-empty History

Figure 10 – Example of production rules for a Begin Operation.

The last two TM operations of the GTS are **commit** and **abort**. Figure 11 shows an eager versioning commit and a lazy versioning commit rule. These rules are executed only when the transaction can in fact commit, otherwise an abort production rule would execute and deal with rollback, which can be specially difficult in an eager versioning algorithm.



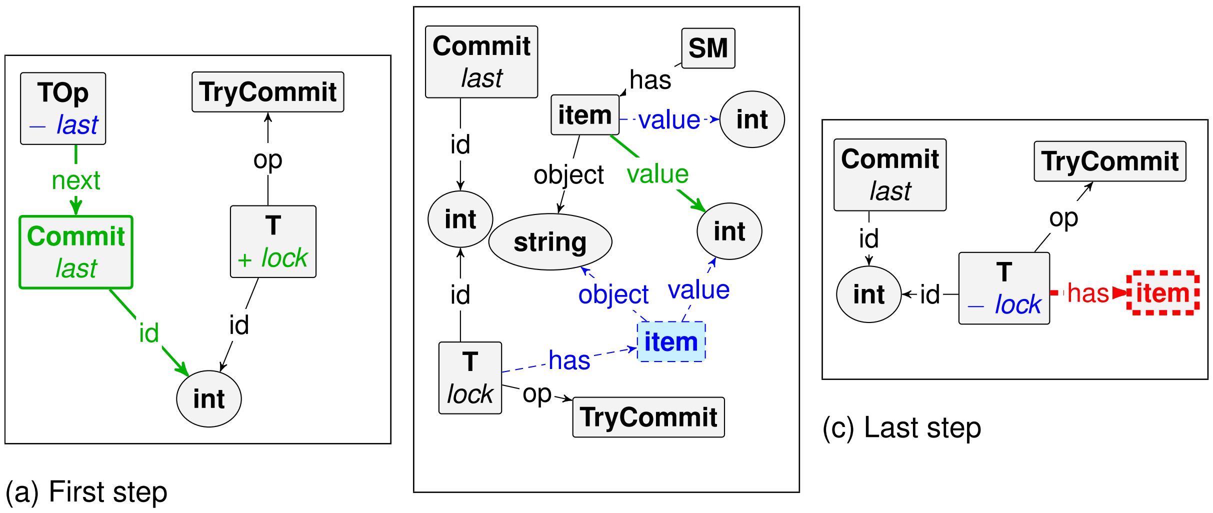
(a) Eager versioning



(a) Lazy versioning

Figure 11 – Example of production rules for a Commit Operation using GROOVE.

In the lazy versioning of a commit seen above a quantifier *forall* (∀) is used to deal with multiple items that store the local values of the variables manipulated. This is a feature of GROOVE, so if the choice is made to not use it the solution would be to split the operation in three production rules as seen in Figure 12. Instead of a single step to deal with a lazy versioning commit, it would require at least two or three, but possibly more. The first step is to lock the transaction in a commit (Figure 12(a), which executes only once), the next step is to apply the local changes to the shared memory (Figure 12(b), executes as many times as there are local items), lastly is to finish the commit and unlock the transaction (Figure 12(c), which executes only once when there are no more local items).



(a) First step

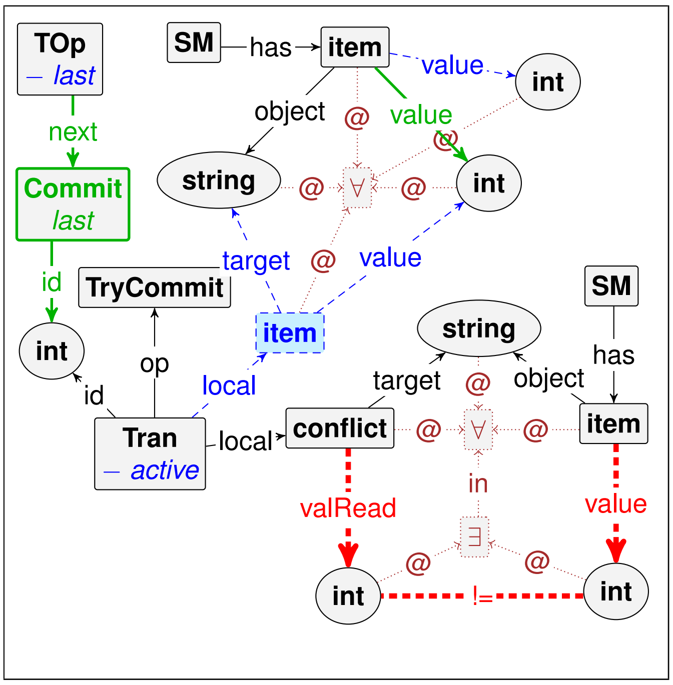
(b) Repeatable step

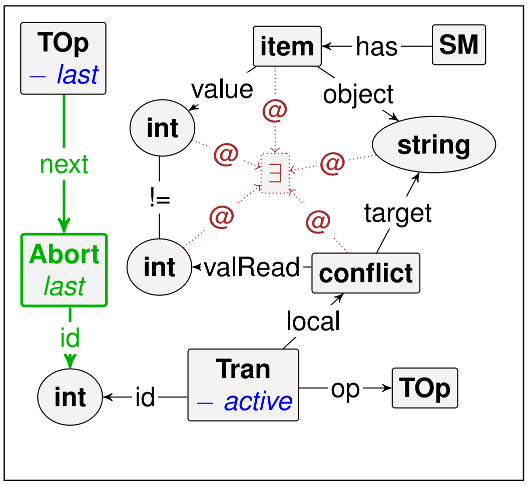
(c) Last step

Figure 12 – Example of production rule for a lazy-versioning Commit Operation divided in steps.

Note that so far only versioning was covered in the production rules, but conflict detection is also an important characteristic to take into consideration when designing a TM algorithm so it will be reflected in the GTS. In an algorithm with eager conflict detection, at any point if a conflict happens some transaction is likely to be aborted. This can approached by always checking the version of a variable read by the transaction. As shown in Figure 13, a *local* node `conflict` stores the value read of each variable the transaction performed a read operation on, this can use that as validation that the transaction has read a stable state of the system. Figure 13(a) shows a commit operation but the same verification happens in all other operations as well. This verification can be read as: for all local nodes `conflict` that store a value read (valRead-edge), their respective objects in the shared memory (`SM` node) must not have a different value.

While the verification of conflict in a commit, read or write operation ensures that all values read are stable, in the abort operation seen in Figure 13(b) by using the





(a) Abort Operation

(a) Commit Operation

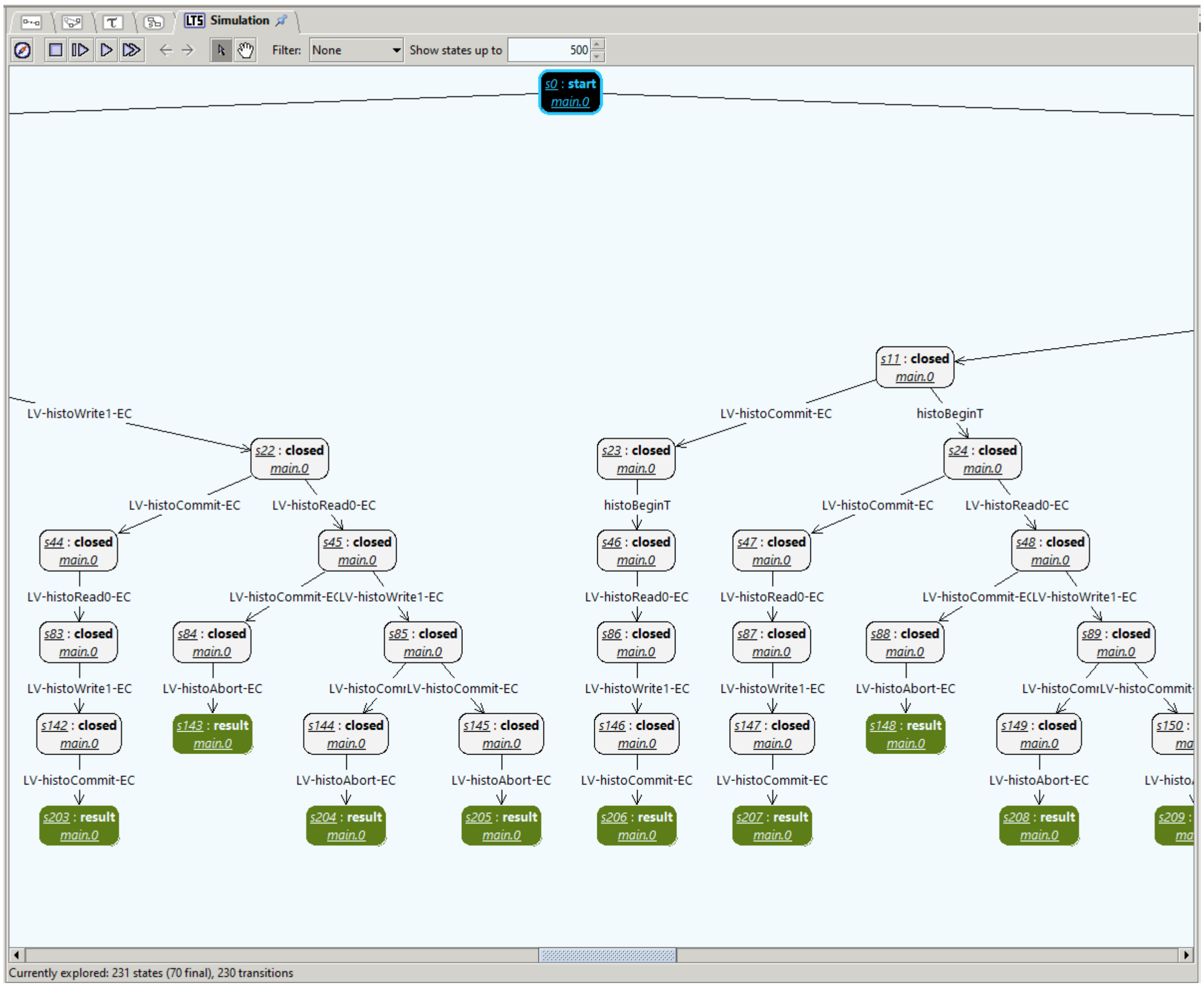
Figure 13 – Example of production rules for a Commit and Abort with Lazy Versioning and Eager Conflict Detection.

quantifier *exists* (∃), with at least one conflict the rule can be triggered. Both instances of verification are mutually exclusive, a transaction cannot commit (or read or write for that matter) and abort at the same time.

7.2 Generating Histories

After translating the algorithm to production rules that correctly modify the state of the system and makes the decision of committing or aborting a transaction, the next step is deal with all possible sequences of operations that generate different histories. Because production rules are being used as a one-step operation that state of the graph, it is possible to use the LTS Simulation tool that GROOVE offers. Given the initial state seen in Figure 4 (in addition of the global nodes `History` and `SM`) the simulation of a *lazy-versioning* and *eager-conflict* algorithm will generate a LTS with 231 states where 70 of those are called “final”. In GROOVE the LTS is visualized with a tree-like graph that can be partially seen in Figure 14.

Each node in the LTS can be expanded (by clicking on it) to visualize the current state of the system resulted from the sequence of production rules applied to that particular state up to that point. At the top of the LTS the initial state labelled *start* can be seen, and at the bottom the final states as green nodes labelled *result*. A final state just means that no more production rules can be applied to that state, this means that there are no more transactional operations are left to be executed and the history generated by that sequence of operations is complete.


Filter: None Show states up to 500

Currently explored: 231 states (70 final), 230 transitions

Figure 14 – Laballed Transition System Simulation in the GROOVE tool.

Those green final nodes are the target of the LTS in this thesis proposal. The LTS demonstrated in Figure 14 shows that this particular initial state of two conflicting transactions generated 70 unique histories. Another feature of GROOVE that can be applied to the generated LTS is the use of Computation Tree Logic (CTL). CTL allows for the verification of a properties in the graphs states in the LTS by using a special production rule called *graph condition*.

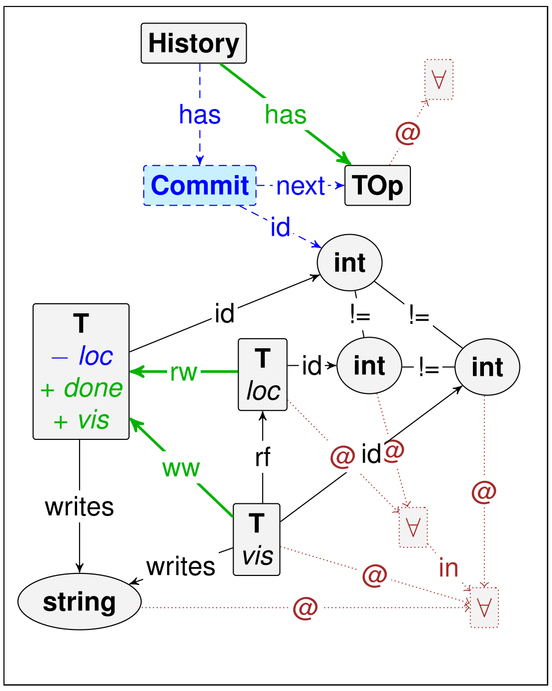
7.3 Correctness Criterion

The correctness criteria used is mainly based on the graph representation of Opacity introduced by Guerraoui; Kapałka (2010). This graph characterization is dependant on a predetermined set of conflicts, and the conflict graph itself is created via the verification of which of these conflicts can be observed between transactions in a history. The LTS is used to explore every combination of transactional operations therefore exploring all possible histories, now the next step is to analyse each of these histories and create their respective conflict graph.

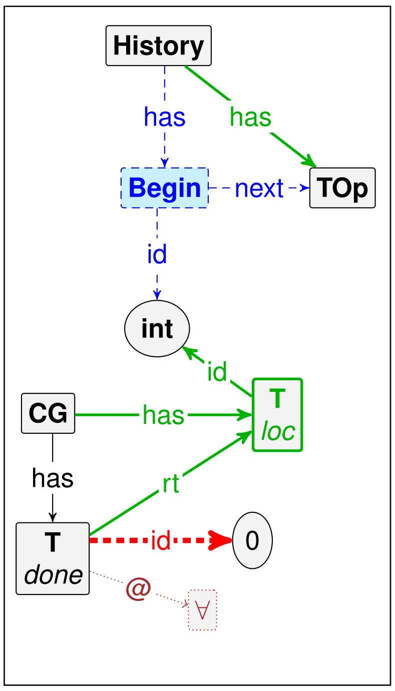
As seen in Definition 4, the conflict graph that represents the graph characterization of opacity contains nodes that represents each transaction in the history (including T_0) and the following edges: real-time (*rt*), reads-from (*rf*), write-after-write (*ww*) and read-before-write (*rw*).

Using the same principles applied in a previous paper (CARDOSO; FOSS; BOIS, 2019), where the opacity of a single history was observed one at a time using a GTS, now the proposed approach was able to analyse the entire LTS constructed above. The process of creating a conflict graph can be separated from the creation of the history itself. Moreover, because it deals with already existing data it only needs to observe the set of conflicts and modify edges between `T`-nodes, that represent each transaction, which results in very simple production rules.

Figure 15 shows the production rules for a commit and a begin operation for the conflict graph of the history being evaluated. Note that, before, a node `Tran` with an *op*-edge was used to identify the current operation of the various parallel transactions, but now because there is only one history being evaluated the node `History` itself is used to path through the sequence. The *has*-edge on the node `History` always starts pointing to the first element of the history. The two operations in Figure 15 cover three out of four conflicts defined by Guerraoui; Kapałka (2010), the remaining one (reads-from) is processed by the read operation which is very simple and similar to the begin operation.



(a) Commit Operation

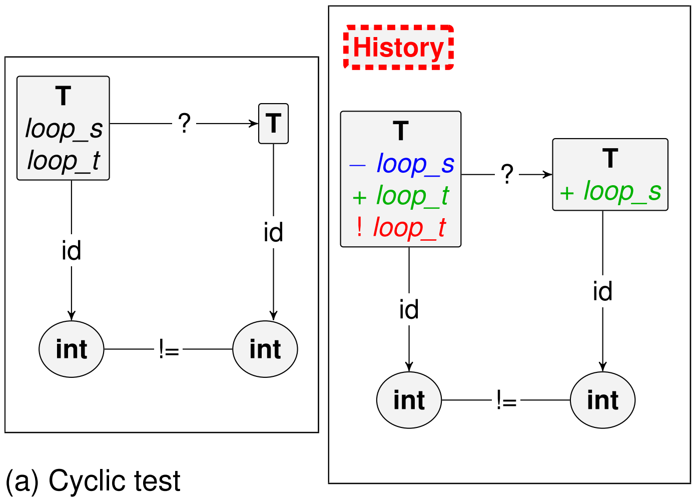


(b) Begin Operation

Figure 15 – Example of production rules for the Conflict Graph Commit and Begin operation.

Lastly, now that the histories were generated, and from each one a conflict graph was extracted, the LTS is complete allowing the use of the Computation Tree Logic (CTL) tool in GROOVE to check for acyclicity of these conflict graphs. With some auxiliary production rules to path through the conflict graphs, a condition is looked for where following the direction of the edges results in a path that goes back to a node that was already visited. The CTL check consists of a pattern match test of a special production rules named *graph condition*: a rule that does not create or delete anything, therefore it does not change the state of the system.

Figure 16 shows the graph condition used in the CTL and the production rule that marks the CG looking for cycles.



(a) Cyclic test

(b) Loop pathing in the CG

Figure 16 – Production rule for the test of acyclicity in a conflict graph.

The verification is made by the formula: `AG !cyclic`, which means that, for every path following the current state, the entire path holds the property of not pattern matching the graph condition `cyclic`. If the formula holds for all states in the LTS, the condition of acyclicity is true and the algorithm only generated opaque histories.

8 CASE STUDY

This chapter describes the main case study derived from this thesis proposal.

8.1 CaPR+

The TM algorithm chosen for this case study is called CaPR+, proposed by Anand; Shyamasundar; Peri (2016), and this section describes the approach to translate the algorithm into a Graph Transformation System[1]. The choice is supported by the fact that this algorithm has a more complex logic, which shows that the approach can deal with more specific and optimized TM algorithms. CaPR+ is an Automatic Checkpoint and Partial Rollback algorithm for Software TM that is based on continuous conflict detection, lazy versioning with automatic checkpointing and partial rollback. In their work, the authors provide a proof of correctness for CaPR+, in particular, Opacity.

The data structures used in the CaPR+ algorithm are categorized into local workspace and global workspace, depending on whether the data structure is visible to the local transaction or every transaction. The data structures used in the local workspace are as follows:

- Local Data Block (LDB): Each entry consists of the local object and its current value in the transaction;
- Shared object Store (SOS): Each entry stores the address of the shared object, its value, a read flag and write flag. Both read and write flags have false as initial value. Value true in read/write flag indicates the object has been read/written.
- Checkpoint Log (Cplog): Used to partially rollback a transaction, where each entry stores, a) the shared object whose read initiated the log entry (this entry is made every time a shared object is read for the first time by a transaction), b) program location from where a transaction should proceed after a rollback, and c) the current snapshot of the transaction's local data block and the shared object store.

[1] Full code available in `https://github.com/anonn34968/gtsfortm`

The data structures in the global workspace are:

- Global List of Active Transactions (Actrans): Each entry in this list contains a) a unique transaction identifier, b) a status flag that indicates the status of the transaction, as to whether the transaction is in conflict with any of the committed transactions, and c) a list of all the objects in conflict with the transaction. This list is updated by the committed transactions.
- Shared Memory (SM): Each entry in the shared memory stores a) a shared object, b) its value, and c) an active readers list that stores the transaction IDs of all the transactions reading the shared object.

Figure 17 shows the type graph defined for the CaPR+ algorithm. Technically all objects are global and the production rules will define what can be modified or not, but for clarity a node `GLOBAL` was used to express the objects that every transaction has access at any point. On the left side, the three global objects are the list of active transactions `AcTrans`, the conflict graph `CG` and the shared memory `SM`. The right side has the transactional operations `TOp` with the inheritance relationship, the transaction node `T` and its local objects `SOS` and `Cplog`. It was decided to omit the `LDB` object from the algorithm because that would imply extra operations on local variables that have no impact in dealing with conflict of the shared memory, which is the main goal of the GTS.

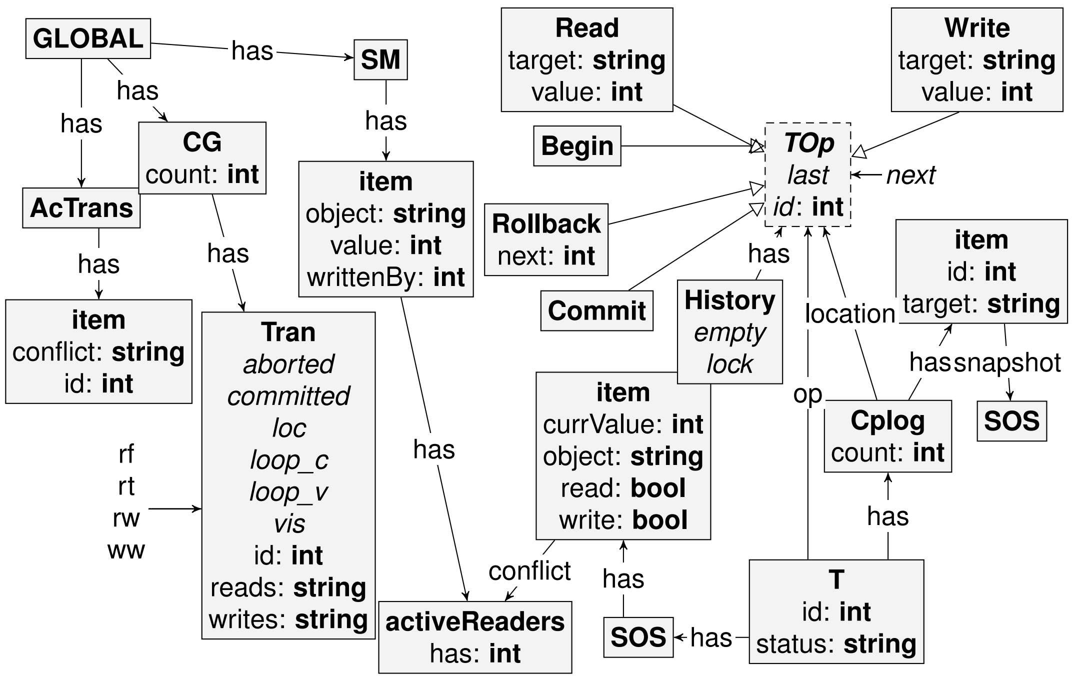


Figure 17 – Type graph of CaPR+ algorithm.

In the CaPR+ algorithm a read operation can be used in two situations: reading a variable from the shared memory for the first time (creating a local copy), and reading it from the local copy if the transaction has one. Figure 18 shows a production rule for a read operation directly to the shared memory, the result of this rule is the creation of the local copy and a checkpoint for a possible partial rollback in the future. In this same rule a node `Read` is added on the last position of the history with the variable name as target and its value from the shared memory. The conflict graph is also modified, with the addition of a relation “reads-from” between the transaction that wrote the value being read and the one executing this read operation.

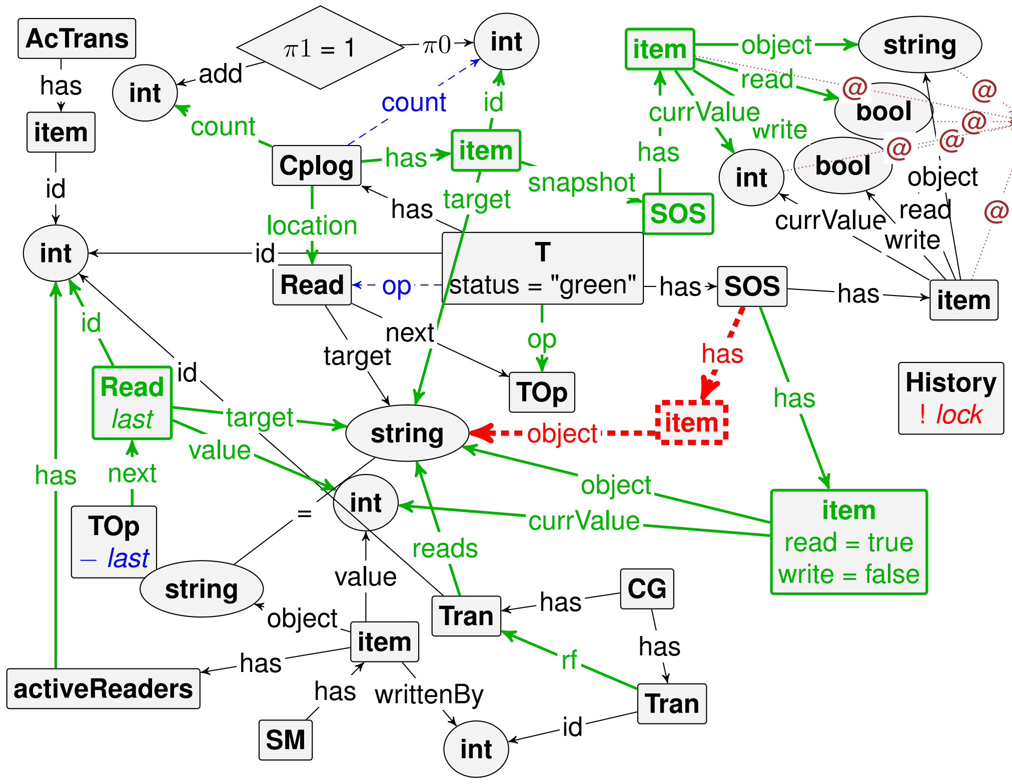


Figure 18 – Read from Shared Memory operation of CaPR+ algorithm.

Another production rule is used to read a variable on a local access, via the SOS object. Any write operation is always done locally. In the original definition by Anand; Shyamasundar; Peri (2016), the algorithm does not have an *abort* procedure, every transaction that tries to commit is either successful or has to rollback to the safest checkpoint (earliest conflicting read operation to the shared memory). Because the algorithm has a lazy versioning characteristic, a few steps need to be taken at commit time. To minimize complexity, the commit production rule was split in different cases, that are mutually exclusive, a commit can be from: a read only transaction (has no conflicts); a write only transaction (has no conflicts); a write only transaction that has conflicts; and a mix of reads and writes that can have conflict.

The way the algorithm deals with conflicts at commit time is by always keeping track of active readers for every variable in the shared memory and, in case of conflict, flagging the respective transaction to be rolled-back later. A simpler algorithm would simply abort the active reader transactions to maintain the correctness of the execution, but CaPR+ tries to always commit all transactions. The result is that some executions will go on for longer where some transactions can even partially rollback multiple times. Correctness is dealt with via the conflict graph of every history, in case of a rollback the transaction is renamed with a new *id* and a new node in the conflict graph is created for the new operations that will be executed.

In this methodology, the conflict based decision making does not care for how many conflicts were detected. During the lifespan of the transaction, every conflict is processed individually, accumulating until the point of the abort/rollback (in CaPR+: either a new read to the TM, or a TryCommit). The existence of a single conflict is enough to flag a transaction as “red”, meaning it needs to be aborted/rolled back. That means that more TM operations will either have no side effects on the conflicts already detected or create more possibilities for the abort operation to happen. In the case of CaPR+, if a transaction T_i commits successfully and flags T_k as “red”, only local reads and writes (which are always local) from T_k would actually execute normally, because any extra read to the TM would trigger the rollback function.

In this proposal, the experiments with different input transactions revealed a relevant point for the demonstration of the algorithm correctness: the ability to observe every type of conflict the criterion defines. The graph characterization of opacity uses mainly conflicts between reads and writes to the same variable, but also has the real-time relation between transactions. It was observed that with three transactions that read and write to the same variable (similar to Fig. 4), it is possible to simulate every conflict in the same LTS. This is relevant because a bigger input would certainly generate more conflicts, which only results in more combinations of TM operations that generate conflicts and a bigger conflict graph. It was concluded that this has no influence on how to deal with conflicts because as said before the decision making processes every conflict individually and deals with them all at once when the conditions to a rollback are met. Thus, the correctness result of the histories generated by the algorithm remains unchanged and an input with three conflicting transactions is a minimal amount to express every combination of conflicts.

Using this approach, the translation of the CaPR+ algorithm to GTS managed to generate only acyclic conflict graphs, proving that the algorithm only generates opaque histories.

Lastly, Figure 19 shows the flowchart demonstrating the execution of the GTS when processing an input of transactions. Each node labelled in red represents a production rule, with a total of 19, that modifies the state of the system. An additional rule, called graph condition, is used for the acyclicity test after the LTS is built. In this flowchart, the nodes and paths in blue are the only ones related to rollback, which is the feature exclusive to this algorithm in particular.

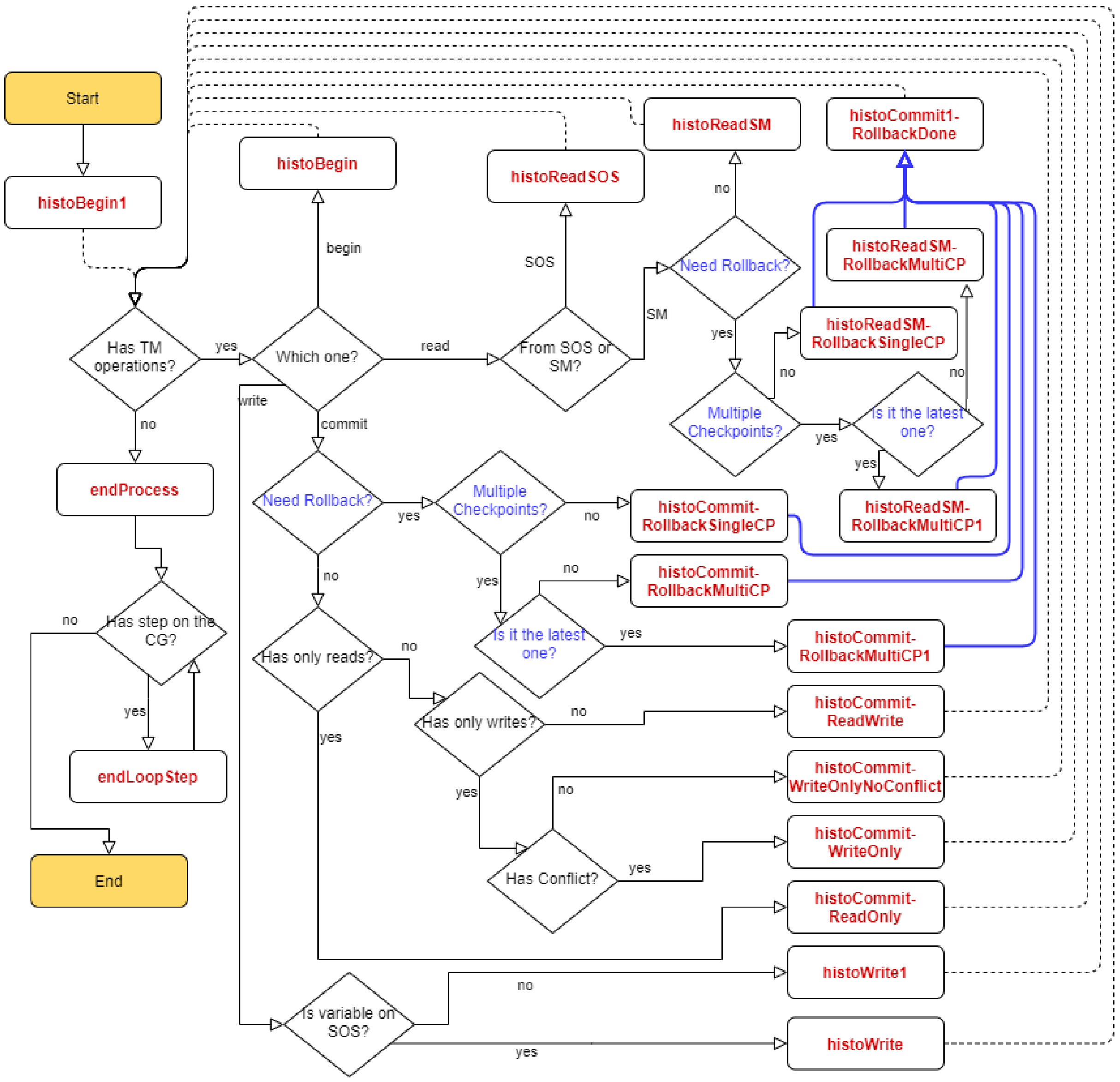


Figure 19 – Flowchart of production rules for the GTS of CaPR+.

REFERENCES

ALPERN, B.; SCHNEIDER, F. B. Defining liveness. **Information processing letters**, [S.l.], v.21, n.4, p.181–185, 1985.

ANAND, A. S.; SHYAMASUNDAR, R.; PERI, S. Opacity proof for CaPR+ algorithm. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING, 17., 2016. **Proceedings...** [S.l.: s.n.], 2016. p.1–4.

ATTIYA, H.; HANS, S.; KUZNETSOV, P.; RAVI, S. Safety and deferred update in transactional memory. In: **Transactional Memory. Foundations, Algorithms, Tools, and Applications**. [S.l.]: Springer, 2015. p.50–71.

BALDAN, P. et al. Towards a notion of transaction in graph rewriting. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.211, p.39–50, 2008.

BARDOHL R.AND MINAS, M.; TAENTZER, G.; SCHURR, A. APPLICATION OF GRAPH TRANSFORMATION TO VISUAL LANGUAGES. **Handbook of graph grammars and computing by graph transformation**, [S.l.], v.2, p.105, 1999.

BLUNDELL, C.; LEWIS, E. C.; MARTIN, M. M. Subtleties of transactional memory atomicity semantics. **IEEE Computer Architecture Letters**, [S.l.], v.5, n.2, p.17–17, 2006.

BUSHKOV, V.; DZIUMA, D.; FATOUROU, P.; GUERRAOUI, R. **Snapshot isolation does not scale either**. [S.l.]: Technical Report TR-437, Foundation of Research and Technology–Hellas (FORTH), 2013.

BUSHKOV, V.; DZIUMA, D.; FATOUROU, P.; GUERRAOUI, R. The PCL Theorem: Transactions cannot be Parallel, Consistent, and Live. **Journal of the ACM (JACM)**, [S.l.], v.66, n.1, p.2, 2018.

CARDOSO, D. J.; FOSS, L.; BOIS, A. R. D. A Graph Transformation System formalism for Software Transactional Memory Opacity. In: XXIII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2019. **Proceedings...** [S.l.: s.n.], 2019. p.3–10.

CLEMENTS, A. T. et al. The scalable commutativity rule: designing scalable software for multicore processors. **Communications of the ACM**, [S.l.], v.60, n.8, p.83–90, 2017.

COHEN, A. et al. Verifying correctness of transactional memories. In: FORMAL METHODS IN COMPUTER AIDED DESIGN (FMCAD'07), 2007. **Anais...** [S.l.: s.n.], 2007. p.37–44.

CORRADINI, A. et al. Algebraic approaches to graph transformation–part i: Basic concepts and double pushout approach. In: **Handbook Of Graph Grammars And Computing By Graph Transformation**: Volume 1: Foundations. [S.l.]: World Scientific, 1997. p.163–245.

DAMRON, P. et al. Hybrid transactional memory. In: ACM SIGPLAN NOTICES, 2006. **Anais...** [S.l.: s.n.], 2006. v.41, n.11, p.336–346.

DOHERTY, S.; GROVES, L.; LUCHANGCO, V.; MOIR, M. Towards formally specifying and verifying transactional memory. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.259, p.245–261, 2009.

DOHERTY, S.; GROVES, L.; LUCHANGCO, V.; MOIR, M. Towards formally specifying and verifying transactional memory. **Formal Aspects of Computing**, [S.l.], v.25, n.5, p.769–799, 2013.

DZIUMA, D.; FATOUROU, P.; KANELLOU, E. Consistency for transactional memory computing. In: **Transactional Memory. Foundations, Algorithms, Tools, and Applications**. [S.l.]: Springer, 2015. p.3–31.

EHRIG, H. et al. Algebraic approaches to graph transformation–part II: Single pushout approach and comparison with double pushout approach. In: **Handbook Of Graph Grammars And Computing By Graph Transformation**: Volume 1: Foundations. [S.l.]: World Scientific, 1997. p.247–312.

EHRIG, H.; ROZENBERG, G.; KREOWSKI, H.-J. rg. **Handbook of graph grammars and computing by graph transformation**. [S.l.]: world Scientific, 1999. v.3.

EMMI, M.; MAJUMDAR, R.; MANEVICH, R. Parameterized verification of transactional memories. **ACM Sigplan Notices**, [S.l.], v.45, n.6, p.134–145, 2010.

FLANAGAN, C.; FREUND, S. N.; YI, J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. **ACM SIGPLAN Notices**, [S.l.], v.43, n.6, p.293–303, 2008.

GHAMARIAN, A. H. et al. Modelling and analysis using GROOVE. **International journal on software tools for technology transfer**, [S.l.], v.14, n.1, p.15–40, 2012.

GUERRAOUI, R.; KAPALKA, M. On the correctness of transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.175–184.

GUERRAOUI, R.; KAPAŁKA, M. Principles of transactional memory. **Synthesis Lectures on Distributed Computing**, [S.l.], v.1, n.1, p.1–193, 2010.

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable Memory Transactions. In: TENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2005. p.48–60. (PPoPP '05).

HARTMANIS, A. C. D. H. J.; HENZINGER, T.; LEIGHTON, J. H. N. J. T.; NIVAT, M. **Monographs in Theoretical Computer Science An EATCS Series**. [S.l.]: Springer, 2006.

HERLIHY, M.; MOSS, J. E. B. **Transactional memory**: Architectural support for lock-free data structures. [S.l.]: ACM, 1993. v.21, n.2.

HIRVE, S.; PALMIERI, R.; RAVINDRAN, B. Hipertm: High performance, fault-tolerant TM. **Theoretical Computer Science**, [S.l.], v.688, p.86–102, 2017.

IMBS, D.; RAYNAL, M. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). **Theoretical Computer Science**, [S.l.], v.444, p.113–127, 2012.

KHYZHA, A.; ATTIYA, H.; GOTSMAN, A.; RINETZKY, N. Safe privatization in transactional memory. **ACM SIGPLAN**, [S.l.], v.53, p.233–245, 2018.

KUMAR, P.; PERI, S. Multiversion Conflict Notion for Transactional Memory Systems. **arXiv preprint arXiv:1509.04048**, [S.l.], 2015.

KUMAR, P.; PERI, S.; VIDYASANKAR, K. A timestamp based multi-version stm algorithm. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING, 2014. **Anais...** [S.l.: s.n.], 2014. p.212–226.

KUZNETSOV, P.; PERI, S. Non-interference and local correctness in transactional memory. **Theoretical Computer Science**, [S.l.], v.688, p.103–116, 2017.

LESANI, M.; PALSBERG, J. Decomposing opacity. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, 2014. **Anais...** [S.l.: s.n.], 2014. p.391–405.

LITZ, H.; DIAS, R. J.; CHERITON, D. R. Efficient correction of anomalies in snapshot isolation transactions. **ACM Transactions on Architecture and Code Optimization (TACO)**, [S.l.], v.11, n.4, p.1–24, 2015.

LYNCH, N. A. **Distributed algorithms**. [S.l.]: Elsevier, 1996.

MANOVIT, C. et al. Testing implementations of transactional memory. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 15., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.134–143.

MARIĆ, O. **Formal Verification of Fault-Tolerant Systems**. 2017. Tese (Doutorado em Ciência da Computação) — ETH Zurich.

MATVEEV, A.; SHAVIT, N. Reduced hardware norec: A safe and scalable hybrid TM. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS, 2015. **Anais...** [S.l.: s.n.], 2015. v.43, n.1, p.59–71.

PANKRATIUS, V.; ADL-TABATABAI, A.-R. A study of transactional memory vs. locks in practice. In: ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2011. **Proceedings...** [S.l.: s.n.], 2011. p.43–52.

PAPADIMITRIOU, C. H. The serializability of concurrent database updates. **Journal of the ACM (JACM)**, [S.l.], v.26, n.4, p.631–653, 1979.

PELUSO, S. et al. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2015., 2015. **Proceedings...** [S.l.: s.n.], 2015. p.217–226.

PETERSON, C.; DECHEV, D. A transactional correctness tool for abstract data types. **ACM Transactions on Architecture and Code Optimization (TACO)**, [S.l.], v.14, n.4, p.1–24, 2017.

RAYNAL, M.; THIA-KIME, G.; AHAMAD, M. From serializable to causal transactions for collaborative applications. In: EUROMICRO 97. PROCEEDINGS OF THE 23RD EUROMICRO CONFERENCE: NEW FRONTIERS OF INFORMATION TECHNOLOGY (CAT. NO. 97TB100167), 1997. **Anais...** [S.l.: s.n.], 1997. p.314–321.

RENSINK, A.; DE MOL, M.; ZAMBON, E. **GROOVE GRaphs for Object-Oriented VErification (Version 5.7.4)**. Disponível em: <https://groove.cs.utwente.nl/>.

SHAVIT, N.; TOUITOU, D. Software transactional memory. **Distributed Computing**, [S.l.], v.10, n.2, p.99–116, 1997.

SIEK, K.; WOJCIECHOWSKI, P. T. Zen and the art of concurrency control: an exploration of TM safety property space with early release in mind. **Proc. WTTM**, [S.l.], v.14, 2014.

WAMHOFF, J.-T.; RIEGEL, T.; FETZER, C.; FELBER, P. RobuSTM: A robust software TM. In: SYMP. ON SELF-STABILIZING SYSTEMS, 2010. **Anais...** [S.l.: s.n.], 2010. p.388–404.

SIGNATURES

Diogo João Cardoso
Proponent

Luciana Foss
Advisor