UNIVERSIDADE FEDERAL DE PELOTAS
Technology Development Center
Postgraduate Program in Computing

Thesis

A Graph Grammar Formalism for Software Transactional Memory Correctness

Diogo João Cardoso

Pelotas, 2023

Diogo João Cardoso

A Graph Grammar Formalism for Software Transactional Memory Correctness

Thesis presented to the Postgraduate Program in Computing at the Technology Development Center of the Federal University of Pelotas, as a partial requirement to achieve the PhD degree in Computer Science.

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

Pelotas, 2023

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

C268g Cardoso, Diogo João

A graph grammar formalism for software transactional memory correctness / Diogo João Cardoso ; Luciana Foss, orientadora ; André Rauber Du_Bois, coorientador. — Pelotas, 2023.

138 f. : il.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2023.

1. Graph grammar. 2. Formalism. 3. Software transactional memory. 4. Correctness. I. Foss, Luciana, orient. II. Du_Bois, André Rauber, coorient. III. Título.

CDD : 005

Elaborada por Simone Godinho Maisonave CRB: 10/1733

Diogo João Cardoso

A Graph Grammar Formalism for Software Transactional Memory Correctness

Thesis approved, as a partial requirement, to achieve the PhD degree in Computer Science, Postgraduate Program in Computing, Technology Development Center, Federal University of Pelotas.

Defense Date: April 6th, 2023

Examining Committee:

Prof. Dr. Luciana Foss (advisor)
Phd in Computer Science at Federal University of Rio Grande do Sul, Brazil.

Prof. Dr. Gerson Geraldo Homrich Cavalheiro
PhD in IT Systems and Communications at Grenoble Institute of Technology, France.

Prof. Dra. Juliana Kaizer Vizzotto
Phd in Computer Science at Federal University of Rio Grande do Sul, Brazil.

Prof. Dr. Rodrigo Geraldo Ribeiro
Phd in Computer Science at Federal University of Ouro Preto, Brazil.

ABSTRACT

CARDOSO, Diogo João. **A Graph Grammar Formalism for Software Transactional Memory Correctness**. Advisor: Luciana Foss. 2023. 138 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2023.

With the rising demand of multi-core systems, the usage of concurrent programming has seen a growth lately. However, the development of correct and efficient concurrent programs is notoriously challenging. A Software Transactional Memory system (STM) provides a convenient programming interface for the programmer to access shared memory without worrying about concurrency issues. STM allows developing programs and reasoning about their correctness as if each transaction executed atomically and without interleaving. In order to check the correctness of any STM system, a formal description of the implementation's guarantees is necessary. There are many correctness conditions for transactional memory, many of them involve the notion of a history of operations to the shared memory, some of which use graphs as tools to help formalize the correctness process. The use of graphs in STM formalism usually involves a depiction of dependencies and relations between transactions and their operations. One correctness condition in particular, called Opacity, uses a graph to represent conflicts between transactions in a history, and a method that extrapolates whether or not that history execution is considered correct.

Graphs and graph transformations play a central role in the modeling of software systems. These models are specified via a formalism that uses a high-level abstraction that is independent of the platform, so the focus can be on the concepts rather than the implementation of the system being formalized. In Graph Grammars (GGs), the modification of graphs is specified via graph transformation rules, which schematically define how a state of the system may be transformed into a new state. When formalizing complex systems, GGs can take into account object-orientation, concurrency, mobility, distribution, etc.

This thesis presents methodology to formalize STM algorithms using graph transformations that includes a verification of the correctness of the executions. The goal is not only to describe the algorithm procedures using production rules, but also include the steps necessary to apply the graph characterization of certain correctness criteria. The methodology includes three main steps: a translation of the algorithm into graph grammar, the method to generate histories and the test for correctness. Some case studies are presented, such as the CaPR+ algorithm, which deals with partial rollbacks. Another case study is the STM Haskell algorithm, which is more well known and is used to compare the behavior between different correctness criteria applica-

tion. Finally, an alternative to the tool GROOVE, used for correctness verification, is presented in the form of an Event-B model that includes the same graph formalism of STM, but differs when dealing with the state space and proof of correctness test. Differently from other work that focuses on formalization of TM algorithms, the methodology proposed deals with graphs in every step, not just in the correctness verification. This results in a methodology that can take advantage of tools and features that focuses on graph transformation and possibly be part of a more automated process for future TM formalization.

Keywords: Graph Grammar. Formalism. Software Transactional Memory. Correctness.

RESUMO

CARDOSO, Diogo João. **Um Formalismo de Gramática de Grafos para Corretude de Memória Transacional de Software**. Orientador: Luciana Foss. 2023. 138 f. Tese (Doutorado em Computer Science) – Technology Development Center, Universidade Federal de Pelotas, Pelotas, 2023.

Com a crescente demanda de sistemas multi-core, programação concorrente tem visto um aumento em uso. Porém, desenvolver programas concorrentes que sejam corretos e eficientes é algo notoriamente desafiador. Um sistema de Memória Transacional de Software (MTS) provém uma interface de programação conveniente para o programador acessar a memória compartilhada sem se preocupar com problemas de concorrência. Um MTS permite desenvolver programas e raciocinar sobre sua corretude como se cada transação fosse executada atomicamente e sem intercalação. Para verificar a corretude de qualquer sistema MTS, uma descrição formal das garantias de implementação é necessária. Existem diversas condições de corretude para memória transacional, muitas delas envolvem a noção de *história* de operações sobre a memória compartilhada, além disso, algumas usam grafos como ferramentas para ajudar a formalizar o processo de correção. O uso de grafos no formalismo MTS geralmente envolve uma representação de dependências e relações entre transações e suas operações. Uma condição de corretude em particular, chamada Opacidade, usa um grafo para representar conflitos entre transações em uma história, e um método que extrapola se a execução da história é considerada correta ou não.

Grafos e transformações de grafos desempenham um papel central na modelagem de sistemas de software. Esses modelos são especificados usando um formalismo que utiliza abstrações de alto nível independente da plataforma, de modo que o foco pode estar nos conceitos e não na implementação do sistema que está sendo formalizado. Nas Gramáticas de Grafos (GGs), a modificação de grafos é especificada por meio de regras de transformação de grafos, que definem esquematicamente como um estado do sistema pode ser transformado em um novo estado. Ao formalizar sistemas complexos, os GGs podem levar em consideração orientação a objetos, concorrência, mobilidade, distribuição etc.

Esta tese apresenta uma metodologia para formalizar algoritmos de memória transacional utilizando gramática de grafos e demonstrar a corretude de suas execuções. O objetivo não é apenas descrever os procedimentos do algoritmo usando regras de produção, mas também incluir os passos necessários para aplicar a caracterização do grafo de certos critérios de corretude. A metodologia inclui três passos principais: uma tradução do algoritmo para uma gramática de grafo, o método para gerar his-

tórias e o teste de corretude. Alguns estudos de caso são apresentados, como o algoritmo CaPR+, que trata de *rollbacks* parciais. Outro estudo de caso é o algoritmo STM Haskell, que é mais conhecido e é usado para comparar o comportamento entre diferentes aplicações de critérios de corretude. Por fim, uma alternativa à ferramenta GROOVE, utilizada para verificação de correção, é apresentada na forma de um modelo Event-B que inclui o mesmo formalismo gráfico do MTS, mas difere ao lidar com o espaço de estados e prova de corretude. Diferente de outros trabalhos que focam em formalização de algoritmos de MT, a metodologia proposta lida com grafos em todos os passos e não só durante o passao de verificação de corretude. Isso resulta em uma metodologia que pode tirar vantagem de ferramentas e características que focam em transformação de grafo e podem possivelmente fazer parte de um processo mais automatizado para para formalizações de MT futuras.

Palavras-chave: Gramática de Grafos. Formalismo. Memória Transacional de Software. Corretude.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

TM	Transactional Memory
STM	Software Transactional Memory
TSO	Total Store Order
CG	Conflict Graph
GG	Graph Grammars
GTS	Graph Transformation System
LHS	Left-Hand Side
RHS	Right-Hand Side
SPO	Single Pushout
DPO	Double Pushout
NAC	Negative Application Condition
LTS	Labelled Transition System
CTL	Computation Tree Logic

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 for executions of regions of code that access a shared memory. 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. As a correctness criterion, it requires all transactions including aborting ones to agree on a single sequential global ordering of transactions. The definition of opacity by Guerraoui; Kapalka (2008) even allows for some sub-classes: Conflict Opacity (CO-Opacity) (KUZNETSOV; PERI, 2017), which only deals with committed transactions; and Multi-version Conflict Opacity (MVC-Opacity) (KUMAR; PERI, 2015; KUMAR; PERI; VIDYASANKAR, 2014) that introduces the notion of multi-versioned variables to the shared memory.

Graphs are usually used in the context of transactional memory as a representation of a happens-before relation during validation or model checking of correctness (ANJANA et al., 2019; DICKERSON et al., 2017; KUMARI; PERI, 2019; PETERSON; DECHEV, 2017), or a graph characterization of logical dependencies that transactions develop by accessing shared variables (LITZ et al., 2014; ZENG, 2020). 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. This conflict can be derived from operations in the total set of events occurred during the transactions execution, or a more restrict set (only committed transactions, for example). Most conflicts are defined in the context of correctness, so they are often defined as the occurrence of two or more transactions that executed a combination at least one read and one write to the same variable that resulted in inconsistencies in the shared memory. A history, a sequence of transactional events that have access to a shared memory, is considered correct if the conflict graph of its transactions presents no cycles.

1.1 Methodology

This thesis proposes a methodology to define a Graph Grammar (GG) that represents a TM algorithm and demonstrates that, considering the notion of conflict introduced by Guerraoui; Kapalka (2008), the algorithm only generates “correct” histories. As an initial model and proof of concept, a GG 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. Then, this graph grammar was expanded to support an entire TM algorithm that generates a set of histories and perform the same correctness

test. This thesis presents a methodology to translate an STM algorithm into a graph grammar, which includes the correctness test, and some case studies of algorithms to show the capabilities of the methodology. The aim of the graph grammar formalism is to show that every execution of the algorithm observes a correct state of the shared memory. 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 well known tool for visual representation of graph grammars called GROOVE (RENSINK; DE MOL; ZAMBON, 2023), 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 grammar would be represented by a much more extensive definition. Another feature of GROOVE is the use of computation tree logic to test properties of the graph grammar's states, this feature is used to demonstrate the correctness test over the state space generated by the grammar.

Besides the state space exploration by the tool GROOVE, this thesis also presents an alternative for correctness verification via an Event-B model (ABRIAL; HALLERSTEDE, 2007). Event-B is a modelling method for formalising and developing systems whose components can be modelled as discrete transition systems. Models are composed of states, (sets of variables) and events that specify possible changes of values of state variables. The encoding of graph grammars in event-B can be seen as an equivalent to the single-pushout approach to graph grammars (EHRIG et al., 1997), and have taken inspirations on research about logic and graphs (COURCELLE, 1997). Properties about the reachable states of the system can be stated as invariants, using First-Order Logic with Set Theory, and proven using theorem provers available for event-B specifications (RIBEIRO et al., 2010). In this thesis, a model for the correctness verification of STM using the graph grammar encoding in event-B is presented as an alternative to the one using the tool GROOVE. The correctness criterion is applied to each history and the acyclicity of its conflict graph is tested. It's worth noting that the event-B model is just an alternative for the property verification of the tool GROOVE, both still use the graph grammar constructions for histories present in the methodology for the algorithm formalization.

1.2 Scientific contributions

The main contributions of this thesis are:

- A methodology to formalize TM algorithms using a graph grammar. This methodology is capable of dealing with different characteristics of TM algorithms in terms of versioning and conflict detection and can used to generate a state space that contains different order of executions of operations to the shared memory.

- The extension of the proposed graph grammar to allow the use of different correctness criteria. The two criteria used are Opacity and CO-Opacity, however any correctness criteria that has a graph characterization can be adapted to this methodology.
- The formalization of more well known TM algorithms STM Haskell and TL2, that can be used to analyze how the decision making of the algorithm influences the resulting correctness of the executions.
- The formalization of the TM algorithm CaPR+ that deals with partial rollbacks. Demonstrating that given a set initial state, the graph grammar formalization of the algorithm can deal with partial rollbacks and generate correct histories.
- A Computation Tree Logic (CTL) approach to applying the correctness test on the algorithms, a feature included in the tool GROOVE.
- An Event-B approach to applying the correctness test on the algorithms as an alternative to using CTL.

1.3 Structure

The remainder of this text is organized as follows:

- Chapter 2 presents the background knowledge about transactional memory, including basic definitions of a transactional memory system, its properties and how the correctness criteria is defined and applied to the system's executions. These definitions will be implemented in the proposed methodology of this thesis.
- Chapter 3 describes the algebraic foundations for graph transformations and graph grammars. This includes the definition of graphs as states that can be transformed with production rules, the type graph that guarantees the correctness of the transformations, the support for attributes and negative application conditions to improve the expressive power of the graph grammar, and finally, a brief description of the tool GROOVE used to aid in the visualization of the created graphs.
- Chapter 4 presents the methodology of formalization for transactional memory algorithms using graph grammars, the main proposition in this thesis. The methodology includes the steps of: first, translating the algorithm into graph notations; second, generating a set of histories; third, testing the correctness of the histories using computation tree logic, a feature of GROOVE.

- Chapter 5 presents an alternative for the correctness test that uses an event-B model, where instead of relying on the state exploration that CTL requires, event-B uses induction proof to reason about properties of reachable states.
- Chapter 6 includes two applications of the graph formalism and correctness test of the algorithms, using the CTL approach.
- Chapter 7 concludes this thesis, presenting final remarks and enumerating future works.

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 an 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 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 thesis.

Finally, Section 2.4 presents a summary of some related work on TM correctness verification with some examples of other methods and tools. Section 2.5 presents some final remarks for this chapter.

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 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.

The values associated to C_T and A_T will depend on how the TM implementation deals with invocations and responses, semantically these values are only used to indicate the success or failure of the execution of that operation. 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*. This includes the case of a re-execution of T in case it aborted.*

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 (executions interleave). 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 are 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. For every TM history *H*, there is also a partial order $\prec_H$ that represents the *real-time order* of transactions in *H*. For all transactions T_k and T_m in *H*, if T_k is completed and the last event of T_k precedes the first event of T_m in *H*, then $\mathsf{T}_k \prec_H \mathsf{T}_m$.

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 mak-

ing 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' ∈ Complete(H), such that*

- *S preserves $\prec_H$, 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*, of every prefix, be equivalent to a history *H' ∈ Complete(H)*, means that opacity treats every transaction *T ∉ Complete(H)* as aborted. It is possible to call *S* as an opaque serialization of *H*. This definition of opacity is similar to du-opacity (SAFETY AND DEFERRED UPDATE IN TRANSACTIONAL MEMORY, 2015).

2.3.2 Graph Characterization of Opacity

Guerraoui; Kapałka (2010) introduced a graph-based characterization of opacity with the purpose of proving 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 $T_i \xrightarrow{rt} T_k$;*

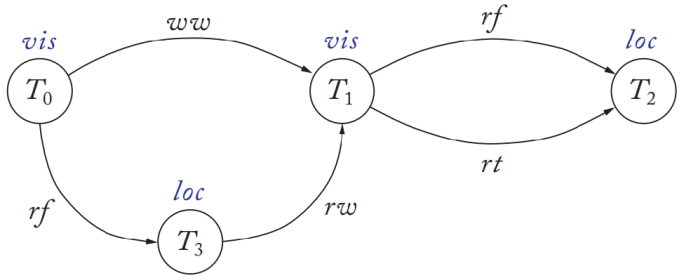
   (b) *If T_k reads from T_i, meaning that T_i writes to the variable before T_k reads it; then the edge is labelled rf and denoted $T_i \xrightarrow{rf} T_k$;*

   (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*.



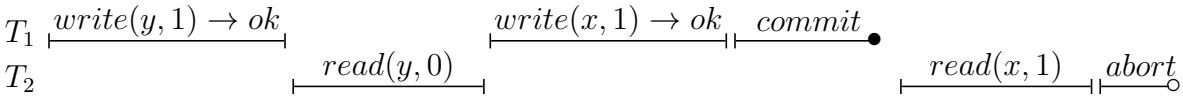
(b) Graph Characterization of history *H*.

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).

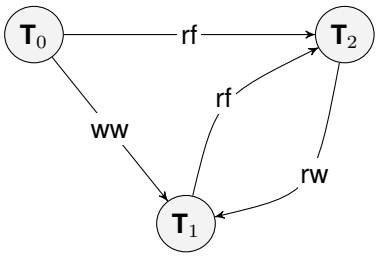
Theorem 1 (Graph characterization of Opacity (GUERRAOUI; KAPAŁKA, 2010))**.** *Any history H with unique writes is final-state opaque if, and only if, exists a version order function $V_{\ll}$ in H such that the graph OPG(H,$V_{\ll}$) is acyclic.*

Proof. Proof can be found in Guerraoui; Kapałka (2010). □

Having this definition of acyclicity of the conflict graph to demonstrate opacity of the history, it is possible to observe that history H in Figure 1 is opaque. An example where the correctness is not satisfied can be seen in history H' in Figure 2(a), where two transactions manipulate t-variables x and y. H' does not have a sequential history that satisfy Definitions 2 and 3, because neither $T_0T_1T_2$ nor $T_0T_2T_1$ have legal read operations. The graph characterization of H' can be seen in Figure 2(b), where a cycle happens between T_1 and T_2.



(a) A history *H'*.



(b) Graph Characterization of history *H'*.

Figure 2 – A non-opaque history (a) and its graph OPG(H',$V_{\ll}$) (b).

2.4 Related works on TM correctness

Several research work has been done on the correctness verification of transactional memory, some approaches (EMMI; MAJUMDAR; MANEVICH, 2010; FLANAGAN; 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 highlight 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 used 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.

2.5 Final Remarks

Transactional memory provides to the programmer a high level abstraction for concurrency control in executions of critical blocks of code. This abstraction is implemented in the form of transactions that enable processes to communicate and synchronize their actions as if they happened in an indivisible and instantaneous point of execution. The executions performed to the shared memory can be logged in what is called a history. Histories generated by a TM algorithm may be evaluated to have properties that reflect the behavior of the algorithm, these properties are usually related to the sequence of operations, the completion or not of transactions, and the legality of reads performed during the execution the history represents.

A correctness criteria is an important aspect of transactional memories as it relates to the consistency of state between the shared memory and the transactions being executed. From the programmer's perspective, transactions look similar to critical sections protected by a global lock where transactions seem to execute sequentially and any aborted transaction is entirely rolled back. The use of a correctness criteria helps with such requirements. Of the several correctness criteria proposed, opacity is highlighted for this thesis. When an execution satisfies opacity, the property guarantees that every transaction observes a consistent state of the shared memory, including aborted ones. The definition of opacity includes a graph characterization that can be used to evaluate the correctness of a given history. This graph, called conflict graph, represents the conflict relations between transactions and its acyclicity represents the satisfaction of the correctness criterion.

This thesis presents a methodology for correctness verification of TM different from previous approaches in the sense that graphs are not just a small part of the formalization, but used instead in the whole process of verification. By developing a formalization that is based on graph transformation, different levels of correctness can be used in the verification of an TM algorithm, based on graph characterization of opacity. This is specially promising for new algorithms that want to include correctness in their definition, or even existing algorithms with new features that need to tested on how they influence correctness of executions. A methodology that uses graphs in the correctness verification of transactional memory executions can potentially be expanded into a tool that automates the whole process, which is something that has yet to be explored in depth.

3 GRAPH TRANSFORMATION

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 (GGs) and Graph Transformation Systems (GTSs), 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. 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. 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).

Graph transformation is 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). GGs 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.

There exist various approaches to realize graph transformations. The two notable algebraic 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. In this work, the SPO approach is adopted, therefore all the definitions presented follow this approach.

The next sections present the fundamentals of graph transformations. Section 3.1 lays the algebraic foundation. It introduces the notions of graphs, morphisms and transformation. In the context of this thesis, the standard definition of graphs are not enough to express the complexities that will be required, so this section also introduces the notion of attributes in graphs and negative application conditions in the transformation rules. Section 3.2 shows the basis for visual representation of graphs used in this thesis. Section 3.3 presents some final remarks for this chapter.

3.1 Algebraic Foundations of Graph Grammars

This section presents the basic concepts of the algebraic single pushout approach as described by (EHRIG et al., 1997). A graph is a structure that represents a set of objects along with relations between them. For the remainder of the text, 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.

Definition 6 (Total Graph Morphism)**.** *A total graph morphism $f : G \to H$ between two graphs is a pair of total functions $f = (f_E, f_V)$ with $f_E : E_G \to E_H$ and $f_V : V_G \to V_H$ 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.*

Definition 7 (Subgraph)**.** *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.*

Definition 8 ((Partial) Graph Morphism)**.** *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 $g' : G \to H$, 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 graphs over a fixed labeling alphabet and the partial morphisms among them form a category denoted by **Graph^P**.

A typed graph is defined by two graphs together with a typing morphism. A typed graph morphism between graphs typed over the same graph is a graph morphism.

Definition 9 (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 $type : G \to TG$.*

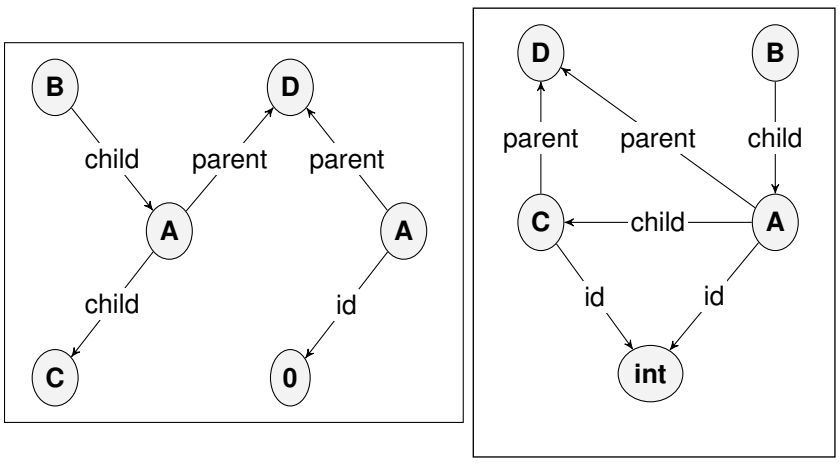
Definition 10 ((Partial) Typed Graph Morphism)**.** *Given a type graph T. Let $G^T = (G, tG)$ and $H^T = (H, tH)$ be typed graphs, where tG and tH are typing morphisms from G and H to T. A **(partial) typed graph morphism** from G^T to H^T is defined by a graph morphism $g = (g_V, g_E)$ from G to H, such that the typed morphism compatibility condition is satisfied:*

$\forall v \in dom(g_V) \cdot tG_V(v) = tH_V(g_V(v))$ *and*

$\forall e \in dom(g_E) \cdot tG_E(e) = tH_E(g_E(e))$

The category of typed graphs and partial typed graph morphisms is denoted by **TGraphP(T)**, where composition and identities are defined componentwise. The full subcategory containing all typed graphs and total morphisms is called **TGraph(T)**.

Figure 3 shows an example of a graph and its respective type graph. In this example, the *A*, *B* and *C*-nodes may be connected by *child*-edges, always in one specific direction (BAC), the node *D* may have a *parent*-edge connected from *A* or *C*, which are the only nodes that may have an *id* connected to an integer value. This *id* is in fact an attribute which will be introduced in the following part of this chapter.



(a) Graph *G*.

(b) Type Graph *TG*.

Figure 3 – Example of graph and its respective type graph.

When dealing with typed attributed graphs, the following definitions are used.

Algebraic specifications may be used to define abstract data types, and algebras to describe the values that may be used as attributes. To include these attributes in a graph the following basic concepts are needed. A *signature SIG* = (*S*, *OP*) consists

of a set *S* of sorts and a set *OP* of constants and operations symbols. Given a set of variables *X* (of sorts in *S*), the set of terms over *SIG* is denoted by $T_{OP}(X)$ (this is defined inductively by stating that all variables and constants are terms, and then all possible applications of operation symbols in *OP* to existing terms are also terms). An *equation* is a pair of terms (*t1*, *t2*), and is usually denoted by *t1 = t2*. A *specification* is a pair *SPEC* = (*SIG*, *Eqns*) consisting of a signature and a set of equations over this signature. An *algebra* for specification *SPEC*, or *SPEC*-algebra, consists of one set for each sort symbol of *SIG*, called *carrier set*, and one function for each operation symbol of *SIG* such that all equations in *Eqns* are satisfied (satisfaction of one equation is checked by substituting all variables in the equation by values of corresponding carrier sets and verifying whether the equality holds, for all possible substitutions). Given two *SPEC*-algebras, a homomorphism between them is a set of functions mapping corresponding carrier sets that are compatible with all functions of the algebras. The set obtained by the disjoint union of all carrier sets of algebra *A* is denoted by $\mathcal{U}(A)$.

Definition 11 (Attributed Graph)**.** *Given a specification SPEC, an* ***attributed graph*** *is a tuple AG = (G, A, AttrG, valG, attrvG) where G = (V_G, E_G, src_G, tgt_G) is a directed graph, A is a SPEC-algebra, AttrG is a set, called* ***set of attributes****, and*

valG : AttrG $\rightarrow \mathcal{U}(A)$*, attrvG : attrG* $\rightarrow V_G$

are total functions. Elements of AttrG are called ***attribute edges****, or just* ***attributes****.*

Definition 12 ((Partial) Attributed Graph Morphism)**.** *A* ***(partial) attributed graph morphism*** g *between attributed graphs AG and AH is a triple* $g = (g_G, g_{Alg}, g_A)$ *consisting of a graph morphism* $g_G = (g_V, g_E)$*, an algebra homomorphism* g_{Alg} *and a partial function* g_A *: AttrG* $\rightarrow$ *AttrH between the corresponding components that are compatible with the attribution:*

$\forall a \in dom(g_A) \cdot g_{Alg}(valG(a)) = valH(g_A(a))$ *and*

$g_V(attrvG(a)) = attrvH(g_A(a))$

An attributed graph morphism g *is called total or injective if all components are total or injective, respectively.*

Definition 13 (Attributed type graph, typed attributed graphs, typed attributed graph morphism)**.** *Given a specification SPEC, an* ***attributed type graph*** *is an attributed graph AT = (T, A, AttrT, valT, attrvT) in which all carrier sets of A are singletons.*

A ***typed attributed graph*** *is a tuple* AG^{AT} *= (AG, tAG, AT), where AG is an attributed graph, AT is an attributed type graph and tAG : AG* $\rightarrow$ *AT is a total attributed graph morphism called attributed typing morphism.*

A ***typed attributed graph morphism*** *between graphs* AG^{AT} *and* AH^{AT} *with attributed type graph AT is an attributed graph morphism* g *between AG and AH such that* g *only relates elements of the same type, that is*

$\forall a \in dom(g_A) \cdot tAG_A(a) = tGH_A(g_A(a))$

In the following definitions only typed attributed graphs will be used, so the word “typed” will be omitted.

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 transformation 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.

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. Items that are mapped in both the LHS and RHS are preserved.

Definition 14 (Attributed Graph Transformation Rule)**.** *Given a specification SPEC = (SIG, Eqns). An* attributed graph transformation rule *over SPEC with type AT is a tuple* $p = (L, R, r, X, RuleEqns, \mathcal{N})$*, where*

- *L and R are typed attributed graphs, called* left-hand side *and* right-hand side*;*
- $r : L \to R$ *is a partial injective graph morphism called* rule morphism*;*
- *X is a set of variables over the sorts of SPEC;*
- *RuleEqns is a set of (conditional) equations using terms of* $T_{OP}(X)$*;*
- $\mathcal{N}$ *is a set of Negative Application Conditions (NACs), which are tuples (N,* n*) where N is a graph and* $n : L \to N$ *a total injective morphism.*

Figure 4 shows an example of an attributed graph transformation rule p'. In this example, the left-hand side graph L has an *A*-node connected to a *D*-node by a *parent*-edge and has an attribute *id* with value of 0. The right-hand side graph R also has an *A*-node with a *parent*-edge to a *D*-node, it does not have an attribute but has a new

C-node connected to *A* by a *child*-edge. The morphism r : L $\rightarrow$ R represents the transformation, where the attribute is deleted an the *C*-node is created, along with their respective edges. The NAC of this rule is represented by the graph N and morphism n, and it denotes that besides the same elements present in L, the *A*-node must not have a *C*-node connected by a *child*-edge.

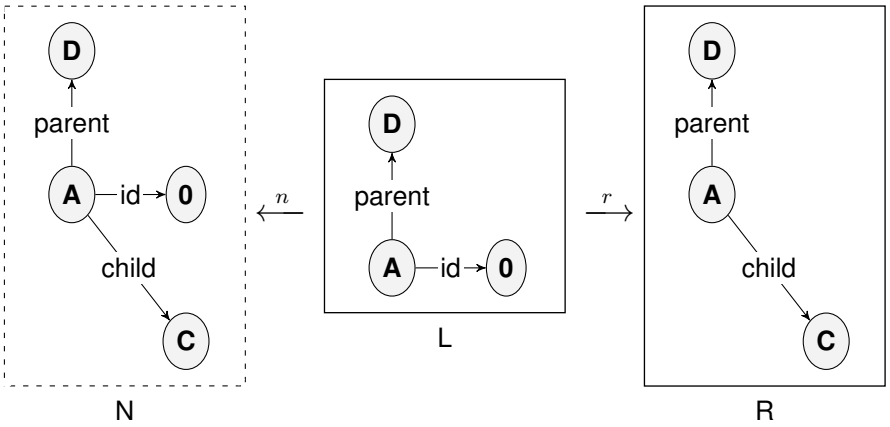


Figure 4 – Graph transformation rule with NACs.

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. In the SPO approach, rules are defined as partial morphism to specify both addition and deletion. Therefore, in addition to the concept of gluing, it has to realize deletion. To restrict the applicability of a rule, a *negative application condition* (NAC) can be used, which forbids specific graph structures from being present in the host graph. Deletion is realized 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 (EHRIG et al., 1997). 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.

A graph grammar is composed of a type graph, an initial graph and a set of rules. To apply a rule in a graph G there must be an insurance that G contains an image of the LHS of the rule and does not contain an image of the forbidden context (described by the NACs of the rule).

Definition 15 (Attributed Graph Grammar)**.** *Given a specification SPEC and a SPEC-algebra A. A* ***(typed) attributed graph grammar*** *is a tuple AGG = (AT, G0, R), such that AT (type graph of the grammar) is an attributed type graph over SPEC, AG0 (initial*

graph of the grammar) is an attributed graph typed over AT using algebra A, and R is a set of rules over SPEC with negative application conditions with type AT.

In order to define a match, it becomes necessary to relate, besides the graph morphism, the variables of the rule to the actual values of carrier sets of the algebra in which the rule will be applied. The match construction must ensure that all equations of the specification and the rule equations are satisfied by the chosen assignment of values to variables. This is achieved by first, lifting the rule to a corresponding one having a quotient term algebra as attribute algebra. This is a standard construction in algebraic specification. Then, the actual match will include an algebra homomorphism from this quotient term algebra to the actual algebra used in the graph to which the rule is being applied. The existence of this homomorphism guarantees that all necessary equations are satisfied.

Definition 16 (Attributed match, NAC satisfaction)**.** *Let a specification SPEC = (SIG, Eqns), a rule with NACs over SPEC r = (AL, AR, r, X, RuleEqns, $\mathcal{N}$) with type AT, where AL = (L, $T_{OP}(X)$, AttrL, valL, attrvL), and a SPEC attributed graph AG typed over AT be given. An **attributed match** $m : \overline{AL} \to AG$ is a total attributed graph morphism m = (m_{Graph}, m_{Alg}, m_A) such that $\overline{AL}$ = (L, $T_{eq}(X)$, AttrL, $\overline{valL}$, attrvL), where $T_{eq}(X)$ is the algebra obtained by constructing the quotient term algebra of the specification (SIG, Eqns $\cup$ RuleEqns) using the set of variables X, and, for all elements $a \in$ AttrlL, $\overline{valL}(a)$ = [valL(a)]. An **attributed match satisfies a NAC** l_j : AL $\to AL^{NAC_j}$, if there is no total attributed graph morphism n : $AL^{NAC_j} \rightarrowtail AG$ that is injective on forbidden items such that $n \circ l_j = m$. An **attributed match satisfies all NACs of a rule** if it satisfies each individual NAC of the rule.*

In practice, given a set of variables *X*, an algebra *A* and defining an evaluation function *eval* : $X \to \mathcal{U}(A)$, there is an unique way to construct the algebra homomorphism (in case it exists). First, all the equations in *Eqns* $\cup$ *RuleEqns* are checked on whether they are satisfied by the assignment. If not, the assignment of values to variables can not lead to an algebra homomorphism, and thus no match can exist using this *eval* function. Otherwise, the extension of *eval* is built to (equivalence classes of) terms, denoted by $\overline{eval}$: $T_{eq}(X) \to \mathcal{U}(A)$.

Definition 17 (Deleted, preserved, created items)**.** *Let r = (L, R, X, RuleEqns, $\mathcal{N}$) be a (typed) attributed rule with left-hand side AL, right-hand side AR and m be a match m = (m_V, m_E, m_{Alg}, m_A) : AL $\to$ AG. The following sets are defined:*

Deleted attributes of AG: *$Del_A^m = m_A[RuleDel_A]$*

Dangling attributes of AG: *$Dangling_A$ = deom(attrvG $\triangleright$ Del_V^m) $\setminus$ Del_A^m*

New attributes of AR: *New_A^m = { $a \in RuleCreate_A$ | m_V(attrvR(a)) $\in$ $Preserv_V^m$ $\vee$ attrvR(a) $\in$ New_V^m }*

Definition 18 ((Attributed) Rule application)**.** *Given a specification SPEC, a rule over SPEC with type T p = (AL, AR, r, X, RuleEqns, $\mathcal{N}$) be a graph transformation rule, where AL = (L, T_{OP}(X), AttrL, valL, attrvL) and AR = (R, T_{OP}(X), AttrR, valR, attrvR) such that AL $\rightarrowtail$ AR, and the total morphism m : (L, T_{eq}(X), AttrL, $\overline{valL}$, attrvL) $\rightarrow$ (G, AlgG, AttrG, valG, attrvG) a match of its LHS L to a graph G that satisfies $\mathcal{N}$. The* ***application*** *of rule p at match m results in the typed attributed graph AH = (H, AlgH, AttrH, valH, attrvH), where*

$$\begin{array}{ccc} L & \longrightarrow & R \\ \downarrow{\scriptstyle m'} & (PO) & \downarrow \\ G & \longrightarrow & H \end{array}$$

H is the resulting graph of applying rule r' = (L, R, $\mathcal{N}$) to graph G, defined by the pushout (PO) over r' and m' = (m_V, m_E) in ***TGraphP(T)****;*

AlgH = AlgG;

AttrH = (AttrG $\setminus$ (Del^m_A $\cup$ Dangling^m_A)) $\uplus$ New^m_A

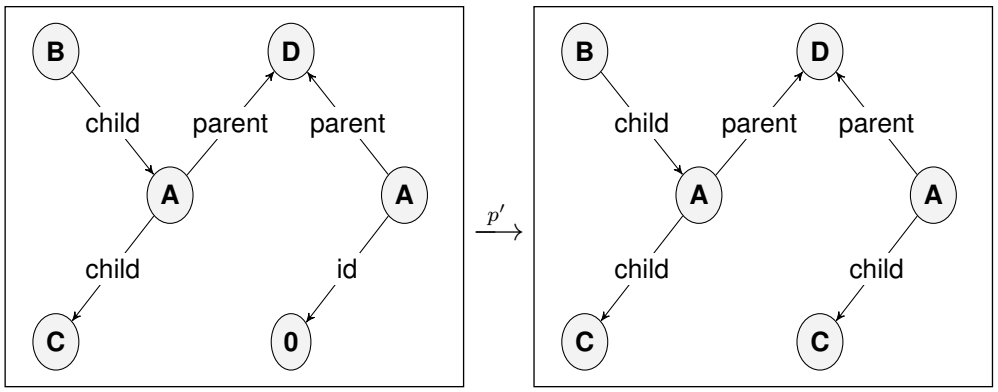
valH = ((Del^m_A $\cup$ Dangling^m_A) $\lhd$ valG) $\cup$ { $a \mapsto$ valR(a) | $a \in$ New^m_A }

attrvH = ((Del^m_A $\cup$ Dangling^m_A) $\lhd$ attrvG) $\cup$

{ $a \mapsto m_v$(attrvR(a)) | $a \in$ New$^m_A \wedge$ attrvR(a) $\in$ RulePreserv$_V$ } $\cup$

{ $a \mapsto$ attrvR(a) | $a \in$ New$^m_A \wedge$ attrvR(a) $\in$ New^m_V }

Figure 5 shows an example of rule application using graph G (Figure 3(a)) and the attributed transformation rule p' (Figure 4). The only possible result from this example is shown in graph H, where an *A*-node with an attribute but no *child*-edge to a *C*-node has its attribute deleted and a *C*-node is created to connect to it using a *child*-edge.



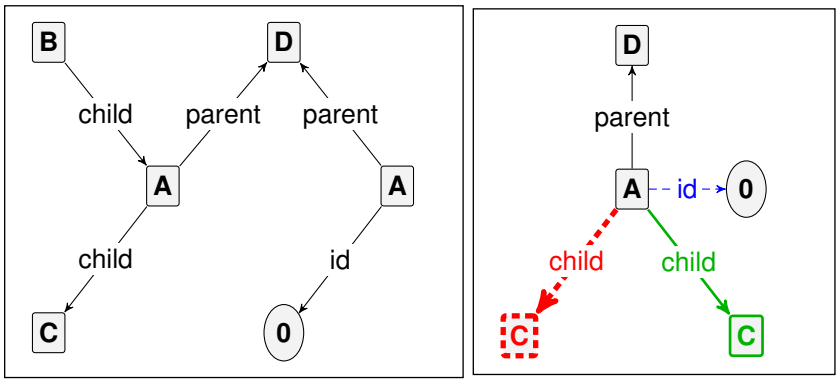
(a) Graph *G*. (b) Graph *H*.

Figure 5 – Example of rule application.

3.2 Visual representation

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, 2023). 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. For the remainder of this work, 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 6 shows the GROOVE representation of graph G and transformation rule p' defined previously. The biggest difference when using GROOVE is in transformation rules, the graph now includes the left and right-hand side and any NAC graph all together. As can be seen in Figure 6(b), the differentiation between the parts of the rule is done via the color and the drawing of the lines. Any element that is to be deleted (present in L but not in R) is drawn in blue dashed lines. Elements that are created (present only in R) are drawn in green. Finally, elements that compose the NACs are drawn in thicker red dashed lines.



(a) Graph *G*. (b) Transformation rule p'.

Figure 6 – Example of graph and transformation rule using GROOVE.

Figure 7 shows a production rule for a transactional begin operation used in this thesis. This rule specifies part of the process of correctness verification that will be described in later chapters. The main goal is to delete the *Begin*-node and at the same time create a *Tran*-node connected to all other preexisting *Tran*-nodes that have an identifier different than zero. Another feature of GROOVE can be seen in this rule, called *flags*. Flags can be interpreted as edges that have source and target as the same vertex. In Figure 7, one *Tran*-node has a flag called “done” (which represents that this specific transaction finished executing), and the other node *T* being created

has a flag of “loc” (meaning the transaction is not visible).

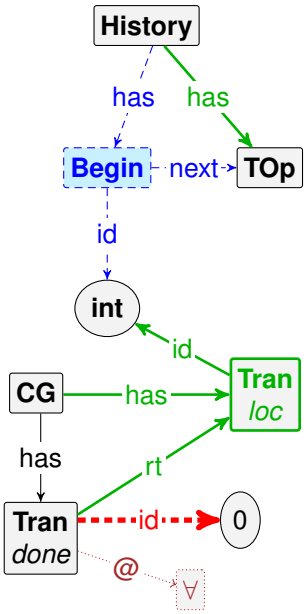


Figure 7 – Example of transformation rule using GROOVE.

The creation of multiple edges is possible due to a feature of GROOVE called quantifiers. In this case, the universal quantifier “for all” ($\forall$) node indicates that an edge *rt* is created with a source for every *Tran*-node with an $id \neq 0$, and the target is always the new *Tran*-node being created. This quantifier can result in an empty set, meaning that even with no *Tran*-nodes with $id \neq 0$ the match can still occur, no *rf*-edges are created as result. Other examples of quantifiers are: the *existential* quantifier ($\exists$), for a mandatory graph pattern; the *non-vacuous universal* quantifier ($\forall^{>0}$), that ensures the match for all occurrences of the graph pattern provided there is at least one; the *optional existential* quantifier ($\exists^{?}$) that tests for an optional existence of a graph pattern. To connect a vertex to quantifiers, GROOVE uses an special edge “at” ($@$). Quantifiers can even be nested by using the “in” edge, allowing for more complex logic for the pattern match and consequences of the rule application.

3.3 Final Remarks

Graph transformation can be seen as the core of most visual languages, and be naturally used to provide a structured representation of a complex system. A graph is a structure that represents a set of objects and the relations between them. If a graph represents a state of a system, transforming this graph (with production rules) can be seen as transforming the state of said system, creating new states. If a type graph is used, there is a guarantee of correctness in the objects being manipulated in the states, and their relations. A typed graph can also be enhanced with the notion of attributes, allowing the states of the system to not only have vertices representing objects but

these vertices may also have attributes that can be typed as integers, strings or even boolean values.

A typed and attributed graph that can be transformed into new graphs has a very high expressive power compared to a standard graph with only vertices and edges. This is most desirable to formalize complex systems that can be seen as multiple objects of different types associated with different values and attributes. In the case of this thesis, a Transactional Memory system. The visual representation of such complex graphs can be aided with the help of a tool like GROOVE that draws objects in a specific way to signify how they behave in the graph transformation.

4 TRANSLATING AN STM ALGORITHM INTO GG

This chapter describes the methodology to formalize a transactional memory algorithm using graph transformation. 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 a graph characterization of a correctness criteria it is possible to use Computation Tree Logic (CTL) model checking to verify that all histories in the LTS satisfy the criterion.

4.1 STM Algorithm

Software Transactional Memory (STM) is a concurrency control mechanism that resolves data conflicts at the software level, as opposed to hardware. STM implementations can be classified using metrics:

- Shared object update (version management): the decision of when a transaction update its shared objects during its lifetime;
- Conflict detection: the decision of when transactions detect a conflict with other transactions in the system;
- Eager or lazy acquisition of objects: a transaction can commit only when all the objects updated by it have been acquired. In this scenario, the acquisition can take place either at the time the object is first accessed (eager), or at commit time (lazy).

Other example of metrics are granularity, visible or invisible reads, lock-based or obstruction-free mechanisms. For this thesis the aspect of versioning and conflict detection were the two metrics taken into consideration. Lock acquisition can also fit in the scope of the model constructed, but because the goal is to evaluate correctness of execution based on conflict, it is assumed that all locks are always acquired successfully. Another reason is the fact that transactional operations are treated as atomic:

the invocation and response always happen subsequently. Therefore, because lock acquisition is part of the internal steps of the operation and in the case of atomic transactional operations would never happens in concurrency with other lock acquisitions, the process is skipped in the graph notation.

The data structures used in the base STM algorithm can be categorized into a local workspace and global workspace. The categorization depends on whether the data structure is visible only to the transaction or to every transaction.

The data structures in the local workspace are:

- Local Variable (LocalVar): each entry represents a local buffer to a variable accessed by the transaction (Figure 8(a)), storing the targeted variable's name, the local value, and two flags for when the local object has been read or written;
- Conflict Checker (Conflict): a list of objects (Figure 8(b)) with information related to the local variables and are used to assist in checking for conflict with the shared memory, each entry stores the targeted variable's name and the value read from the shared memory.

The data structures in the global workspace are:

- Shared Memory (SM): each entry on the shared memory (Figure 9(a)) stores a shared object (transactional variable, TVar) and its value;
- Transactions (T): each transaction (Figure 9(b)) has an unique identifier (ID), a flag for when it is active or not, a list of Conflict objects, and a list of Local Variable objects;
- History: represents the sequence of actions executed to the shared memory, each entry (Figure 9(c)) stores the type of operation, the ID of the transaction that executed it, and some extra information depending on the type of operation;

Target	Value	Read	Written

(a) **Local Variables**: list of **LocalVar**-objects, variables local to a single transaction.

Target	Value Read (ValRead)

(b) **Conflict Checker**: list of **Conflict**-objects used to detect possible read conflicts.

Figure 8 – Local workspace data structures for a TM algorithm.

The description for the various procedures of an **lazy versioning and eager conflict detection** STM algorithm are:

Object	Value

(a) **Shared Memory**: contains a list of **TVar**-objects, transactional variables in the shared memory.

ID	Active	Conflict List (CList)	Local Variables (LVList)

(b) **Transactions**: list of **T**-objects representing transactions.

Operation	Transaction ID	From	Target	Value

(c) **History**: sequence of operations representing a history.

Figure 9 – Global workspace data structures for a TM algorithm.

- ConflictChecker: the conflict check is an auxiliary function that looks at every active transaction's conflict list to see if any value read is different from the shared memory. Any inconsistent value indicates a conflict between different transactions.
- Read: when reading the object labelled *var* the transaction *T* first checks for conflicts (by evoking ConflictChecker), if any are found the transaction is flagged as "inactive", a new entry (*Abort*) is added to the history and the read operation aborts. If no conflicts are found, the transaction looks for *var* in the local variable list (LVList), if found the local variable is flagged as "read" and its value is returned. If *var* is not local, the transaction looks for it in the shared memory. If found, a new local variable object, a new conflict object are created to add to *T*. The new local object is flagged as "read" but not "written" and a new operation object is added to the history. If *var* is neither a local nor shared variable, an error is returned.
- Write: when writing to *var*, the same conflict check is executed. If no conflict exists, the transaction looks for *var* in the local variables to rewrite it and flag it as "written". If *var* is not local and is found in the shared memory, a new local object is created with the value being written. This local variable is flagged as "written" but not "read". Lastly, a new operation is added to the history.
- Commit: to execute a commit there should be no conflicts with the shared objects' values. With no conflicts, for each local variable that the transaction executed a write, their respective shared object is updated with the new value. A new object is added to the history, completing the operation.

For Algorithm 1, seen below, data structures are used in the form of a method call. For example, *SM.TVar(var)* is to be interpreted as finding the *TVar* object in *SM* that

has a *object name* equals to *var*. In this case, *object* is used as the key to find the object (Figure 9(a)), however other data structures may use other keys, the highlighted keyword in the object describing the data structures represents their respective key. In the case of the data structure for the history, no key is needed because the algorithm will never perform a search for a specific object in it. The only operations allowed on histories are *enqueue* and *dequeue*. The main reason to use such form in the operations for an algorithm is because this object oriented-like notation is very easily translated into graph notations. An object *SM* that has many child-objects *TVar* where each *TVar* have an unique key that can be used for a search operation resembles a case where a pattern match is used with vertices with specific edges and attributes.

In summary, the eager conflict detection feature of the algorithm will determine that when an operation is invoked, the first thing that is executed is a verification of any conflict with the shared memory (lines 2, 25 and 45). If a conflict exists the algorithm aborts that current operation that was called, adds a new entry to the history and finishes the execution of that procedure in specific. The lazy versioning side of the algorithm will ensure that a local copy of any transactional variable is created when first accessed (line 11). This local copy takes precedence over the shared memory counterpart for any future access, in a read operation the conditional in line 7 happens before the condition in line 10, same for the write operation in lines 30 and 33.

STM Algorithm 1 - Lazy Versioning and Eager Conflict Detection

```
1:  procedure Read(var, T)
2:      if ConflictChecker() = TRUE then
3:          newOperation ← {op: Abort, tranID: T.ID};
4:          History.enqueue(newOperation);
5:          T.active ← FALSE;
6:          return Abort;
7:      else if var is in T's local variables then
8:          T.LVList.LocalVar(var).reads ← TRUE;
9:          return T.LVList.LocalVar(var).val;
10:     else if var is in shared memory then
11:         newLocalVar ← {target: var, val: SM.TVar(var).val};
12:         newConflict ← {target: var, valRead: SM.TVar(var).val};
13:         newOperation ← {op: Read, tranID: T.ID,
                from: SM.TVar(var).writtenBy, value: SM.TVar(var).val};
14:         T.LVList.LocalVar(var).reads ← TRUE;
15:         T.LVList.LocalVar(var).writes ← FALSE;
16:         T.CList.push(newConflict);
17:         T.LVList.push(newLocalVar);
18:         History.enqueue(newOperation);
19:         return SM.TVar(var).val;
```

```
20:     else
21:         return Error;
22:     end if
23: end procedure
24: procedure Write(var, value, T)
25:     if ConflictChecker() = TRUE then
26:         newOperation ← {op: Abort, tranID: T.ID}
27:         History.enqueue(newOperation)
28:         T.active ← FALSE;
29:         return Abort;
30:     else if var is in T's local variables then
31:         T.LVList.LocalVar(var).writes ← TRUE;
32:         T.LVList.LocalVar(var).val ← value;
33:     else if var is in shared memory then
34:         newLocalVar ← {target: var, val: value};
35:         newOperation ← {op: Write, tranID: T.ID,
                from: SM.TVar(var).writtenBy, value: SM.TVar(var).val};
36:         T.LVList.LocalVar(var).reads ← FALSE;
37:         T.LVList.LocalVar(var).writes ← TRUE;
38:         T.LVList.push(newLocalVar);
39:         History.enqueue(newOperation);
40:     else
41:         return Error;
42:     end if
43: end procedure
44: procedure Commit(T)
45:     if ConflictChecker() = TRUE then
46:         newOperation ← {op: Abort, tranID: T.ID};
47:         History.enqueue(newOperation);
48:         T.active ← FALSE;
49:         return Abort;
50:     end if
51:     for each LocalVar object in T.LVList do
52:         for each TVar object in SM do
53:             if LocalVar.target = TVar.object and LocalVar.writes = TRUE then
54:                 TVar.val ← LocalVar.val
55:                 TVar.writtenBy ← T.ID
56:             end if
57:         end for
58:     end for
59:     newOperation ← {op: Commit, tranID: T.ID}
```

```
60:     History.enqueue(newOperation);
61: end procedure
62: procedure ConflictChecker()
63:     for each active transactions T do
64:         for each Conflict object in T.CList do
65:             if Conflict.valRead ≠ SM.TVar(Conflict.target).val then
66:                 return TRUE;
67:             end if
68:         end for
69:     end for
70:     return FALSE;
71: end procedure
```

For the **eager versioning** algorithm, reads and writes operate directly to the shared memory. To ensure correctness of the execution, the possibility of a rollback in the case of a conflict is required. The implementation of a rollback can be done with an extra data structure (Figure 10) for a log that stores snapshots of the shared memory to rollback to, and an extra procedure to execute the rollback (Algorithm 1). This procedure should not only replace the current shared objects with a correct snapshot, but also make sure to flag any transaction that is aborted as result of the rollback. One way to implement this is to save a snapshot every time a commit is executed successfully, and always keep track of transactions that have actively read variables from the shared memory from that point forwards.

ID	List of TVars	Active Readers

Figure 10 – **Log**: list of snapshots of the shared memory.

STM Algorithm 2 - Eager Versioning's rollback procedure

```
1:  procedure Rollback()
2:      snapshot ← Logs.pop()
3:      for each tID in snapshot.activeReaders then
4:          flag T(tID) for abort
5:      end for
6:      for each TVar in SM then
7:          override TVar with snapshot.TVar
8:      end for
11: end procedure
```

To modify this algorithm into one with **lazy conflict detection**, changing the invocation of the conflict checker to only happen at commit time is enough.

The methodology of translating an STM algorithm into a GG also describes how to execute a correctness verification on histories generated by the algorithm. A conflict graph is used as the data structure to analyse the presence of cycles. Figure 11 shows an example of the data structure for the conflict graph, storing a *Tran*-object for each transaction in the history. The manipulation of these objects and the verification using the values in “Loop Step” and “Loop Token” is what comprises the correctness verification.

ID	Vis	Loc	Done	Loop Step	Loop Token	Reads	Writes

Figure 11 – **Conflict Graph**: list of Tran-objects.

The entire process of correctness verification is made of extra operations that process each operation in a history to build a conflict graph.

4.2 Graph Grammar

First, a representation of sequential operations is needed. This representation will compose transactions and histories used in the remainder of the text. Figure 12 shows an example of two transactions with some conflicting operations, a code like this is the input for the system being evaluated.

$\mathbf{T}_1$	$\mathbf{T}_2$
1: *begin*	1: *begin*
2: *read(x)*	2: *read(x)*
3: *write(x,1)*	3: *write(x,2)*
4: *tryCommit*	4: *tryCommit*

Figure 12 – Example of transaction code.

In Figure 13 it is demonstrated how the code from Figure 12 is represented in a graph manner. This is the **initial state** as an input to the GG. 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. Note that the sequential operations are connected by a directed edge *next* that represents the order in which these operations must execute. The edge *op* points to the current operation to be executed, in an approach like this every transactional operation is considered to be atomic: the response of the operation happens immediately after the invocation. These operations represent the invocation order that each transaction requires, moreover, as this is just an evaluation scenario, the *tryCommit* operation represents an attempt to commit the transaction. The result of the attempt is not known at this point, hence “try”, and will appear only in the history of each execution

that reaches that point of the transaction. In early iterations of the graph formalism described in this chapter, that only dealt with single histories, invocations and responses were processed separately. However, because some of the correctness criteria for the TM algorithms use atomic operations, it was decided that for the full algorithm formalism it is best to keep the atomic notion, which in turns decreases the number of nodes in a transaction or history, making it more readable.

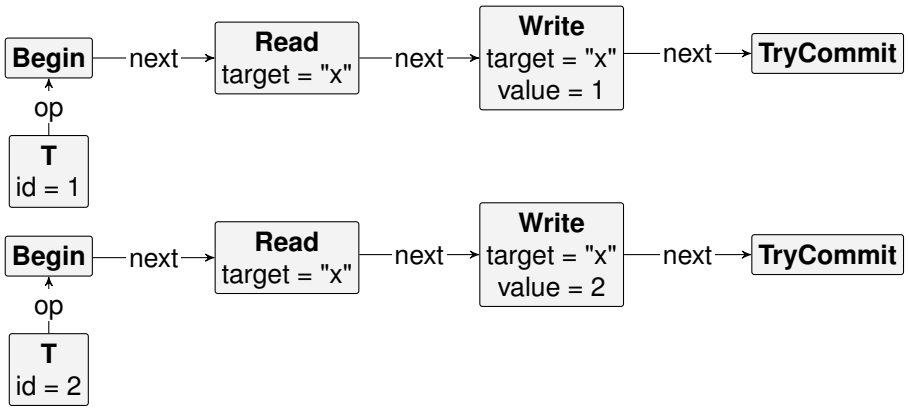


Figure 13 – Graph representation of transactions used as input for the GG.

4.2.1 Initial State and Type Graph

The initial state of the graph grammar includes the transactions (as seen in Figure 13) 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 14 shows some examples of global objects that the algorithm logic will allow to be accessed at any moment. The object *SM* represents the shared memory that has a list of transactional variables (*TVar*), the conflict graph is represented by *CG* and has a list of transactions (*Tran* objects) based on the current execution, and the history starts with a flag of “empty”. This restriction of access to each of these objects is enforced by the transformation rules.

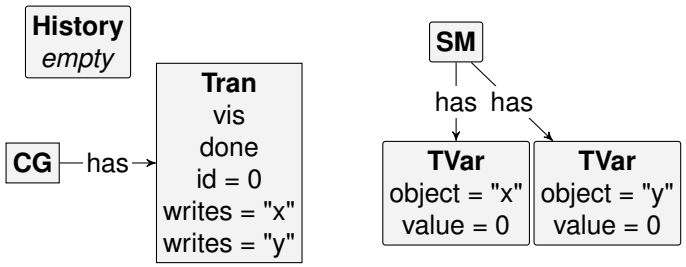


Figure 14 – Graph representation of global objects in the initial state of the GG.

Another important aspect of the graph grammar 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 15 shows an example of a

type graph for 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 supertype 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* (via 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.

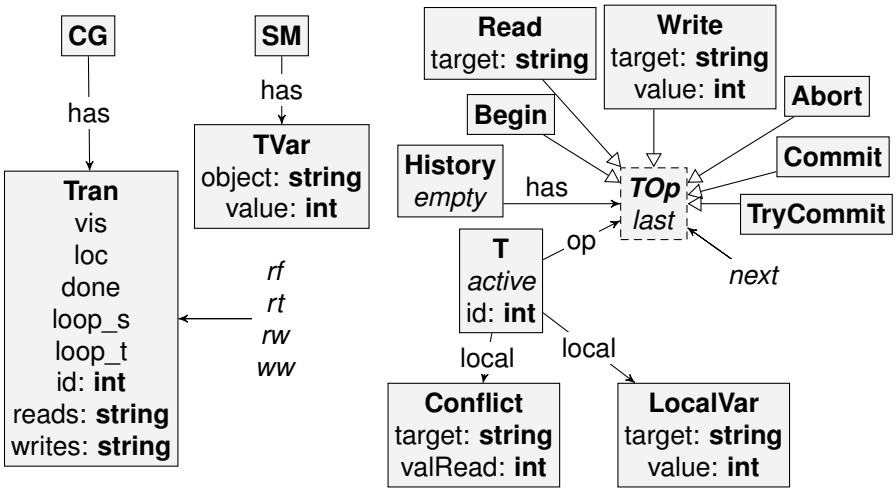


Figure 15 – Example of Type Graph of the GG.

The next step is to define how the execution of these transactions will generate a history. For simplicity it was only considered the four operations mentioned above (begin, read, write and try-commit) and each operation is considered atomic. It is up to the algorithm as to what the side effects of each operations are, but for the sake of the goal of evaluating all histories, every operation is seen as a step to create a new entry to the history. Similar to how a transaction was represented in Figure 13, a history is a sequence of operations connected by a *next*-edge with the corresponding transaction *id* and necessary information of that operation (targeted variable and values that were read or written). Figure 16 shows an example of a history after some transactional operations have been executed. In this history two transaction (with *ids* 1 and 2) execute a sequence of operations to the shared memory and transaction 1 commits whereas transaction 2 aborts.

4.2.2 Production Rules

GROOVE builds a state space in the form of an LTS representing every sequence of application of transformation rules. This LTS is usually represented by a graph where the “root” is the initial state (Figure 13 for example) and each connected node is the state resulted from a rule application. When dealing with STM algorithms, the order of

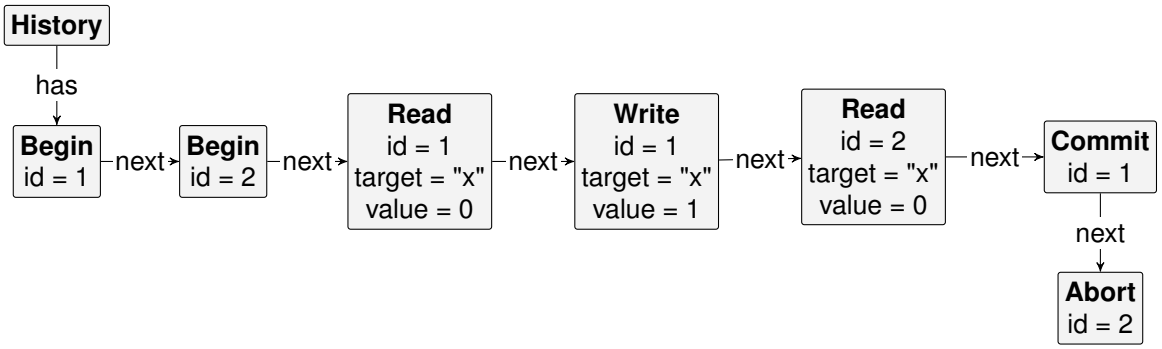
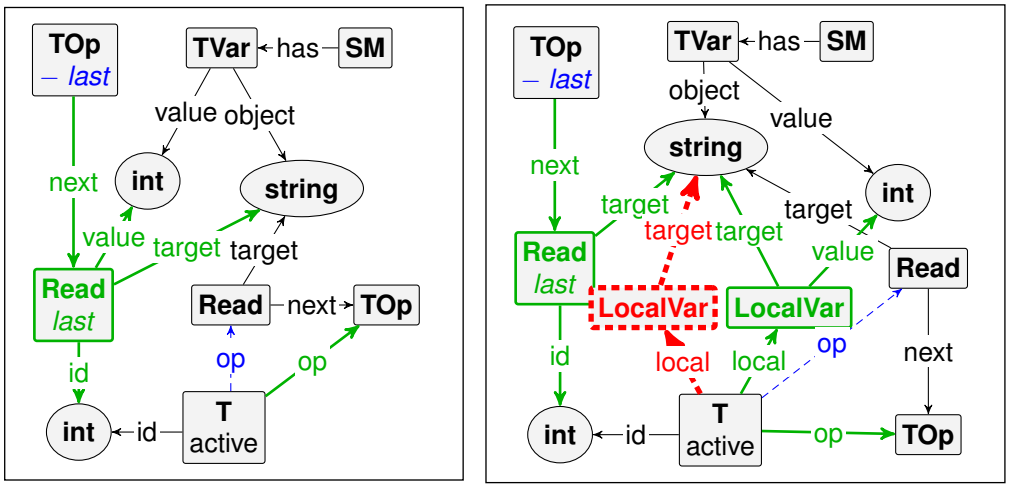


Figure 16 – Example of graph representation of a history.

applications for production rules result in an LTS with the form of a tree-like graph. The final state in the LTS is a state where no rules can be applied via pattern match and therefore they are the leaves of this tree.

Rules ideally consume/process each transactional operation in a single step. At each step, the operation is added to the history and changes are applied to the shared memory accordingly. This means that every final state contains a full history execution of all operations in the initial state. Therefore, because the LTS contains all possible rule application orders, it essentially contains a graph representation of every history of a given set of transactions.

The first operation described is a **read operation**. In Figure 17 it is showcased two different approaches of a read to the shared memory: an eager versioning in Figure 17(a) and a lazy versioning in Figure 17(b).



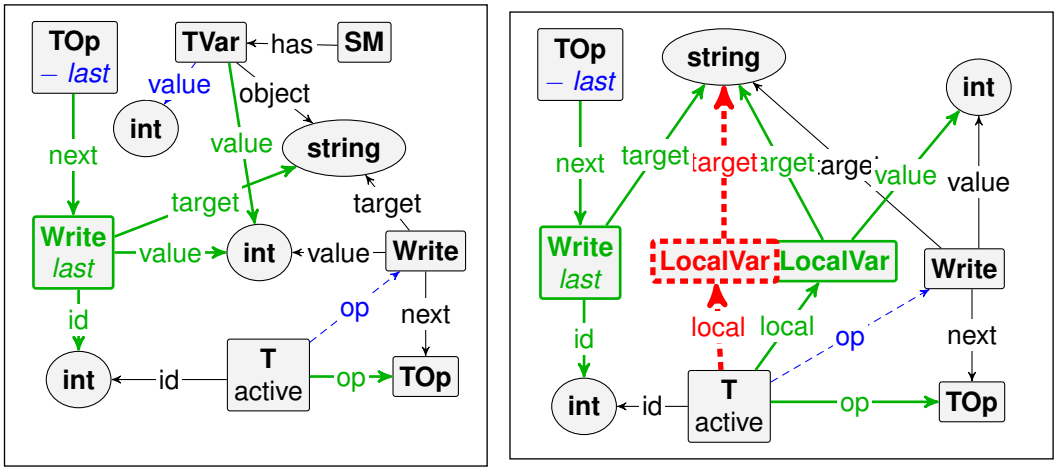
(a) Eager versioning (b) Lazy versioning

Figure 17 – Example of production rules for a Read Operation.

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 variable 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.

The **write operations** are treated in a similar way. Figure 18 shows an eager and a lazy versioning approach to a production rule that executes a write. In same situation of the rules seen above, the lazy versioning write operation also needs an extra production rule to execute the write locally if the local copy already exists.



(a) Eager versioning

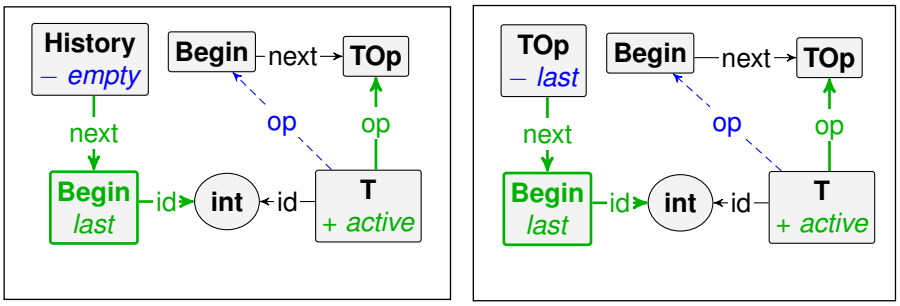
(b) Lazy versioning

Figure 18 – Example of production rules for a Write Operation.

Some similarities can be seen between the production rules shown in Figures 17 and 18: an object is added at the end of the sequence representing the history (*TOp* loses the *last*-flag and points to a newly created *Read*/*Write* node 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 transactional operations (node that *T* points to with an *op*-edge) 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 19(a) and 19(b), and these two are the only rules required to deal with starting transactions.

The last two operations are **commit** and **abort** operations. Similarly to the begin operation these two need more context to be created, more specifically the logic behind the actual algorithm being translated into a graph grammar. The decision to commit a transaction, or not, usually includes conflict detection with other transactions and

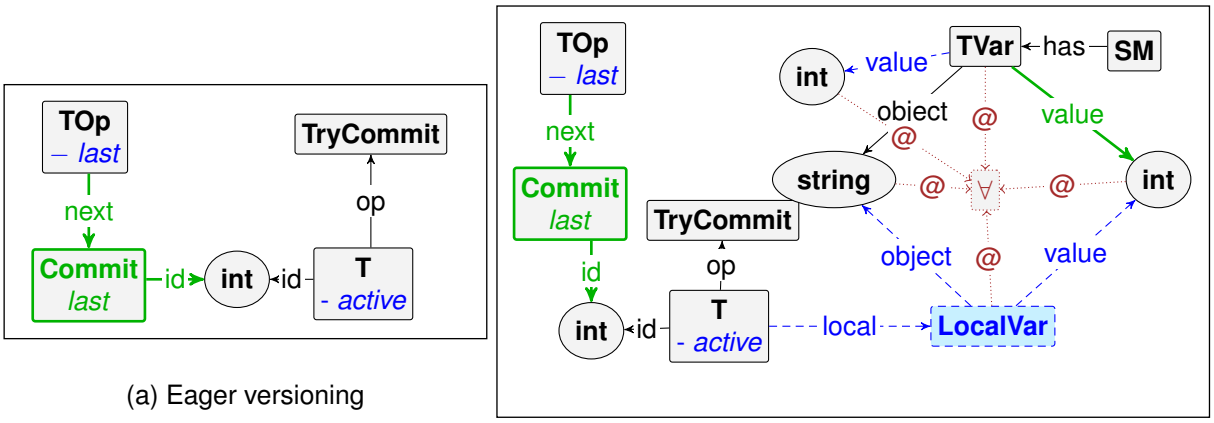


(a) Empty History (b) Non-empty History

Figure 19 – Example of production rules for a Begin Operation.

how to deal with it. A simple example would be a TM algorithm that reads and writes directly to the shared memory but keeps a log of previous values for each t-variable and rolls back any changes if it detects conflicts at any time (eager versioning and eager conflict detection). Another example would be that transactions keep a local copy of the t-variables they read/write and only at the commit operation they verify conflicts and deal with them (lazy versioning and lazy conflict detection). The choice between lazy or eager conflict detection has been found to influence schedulability when implementing real-time TM systems (BELWAL; CHENG, 2011). Both can be translated into the rules of the GG.

Figure 20 shows an eager versioning commit and a lazy versioning commit rule.



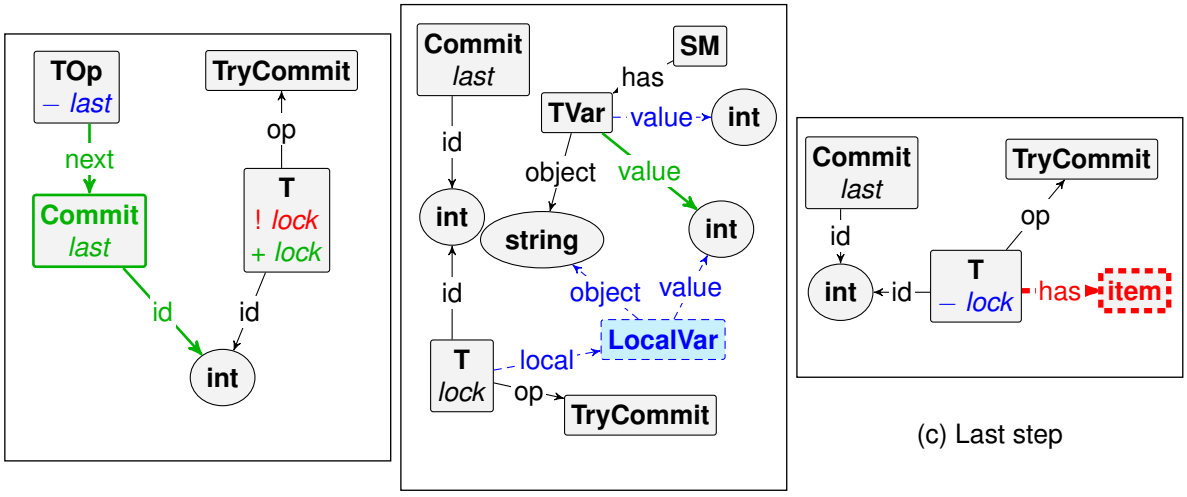
(a) Eager versioning

(a) Lazy versioning

Figure 20 – Example of production rules for a Commit Operation.

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 difficult in an eager versioning algorithm. This example shows the use of a universal quantifier (∀), this makes it so that for every *LocalVar* node in the transaction committing, their values will be written to the corresponding *TVar* node in the shared memory. This is a feature of GROOVE, so if a choice is made to not use it, the solution would be to split the operation in three production rules as seen in Figure 21. 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 21(a), which executes only once), the next step is to apply the local changes to the shared memory (Figure 21(b), executes as many times as there are local variables), lastly is to finish the commit and unlock the transaction (Figure 21(c), which executes only once when there are no more local variables).



(a) First step

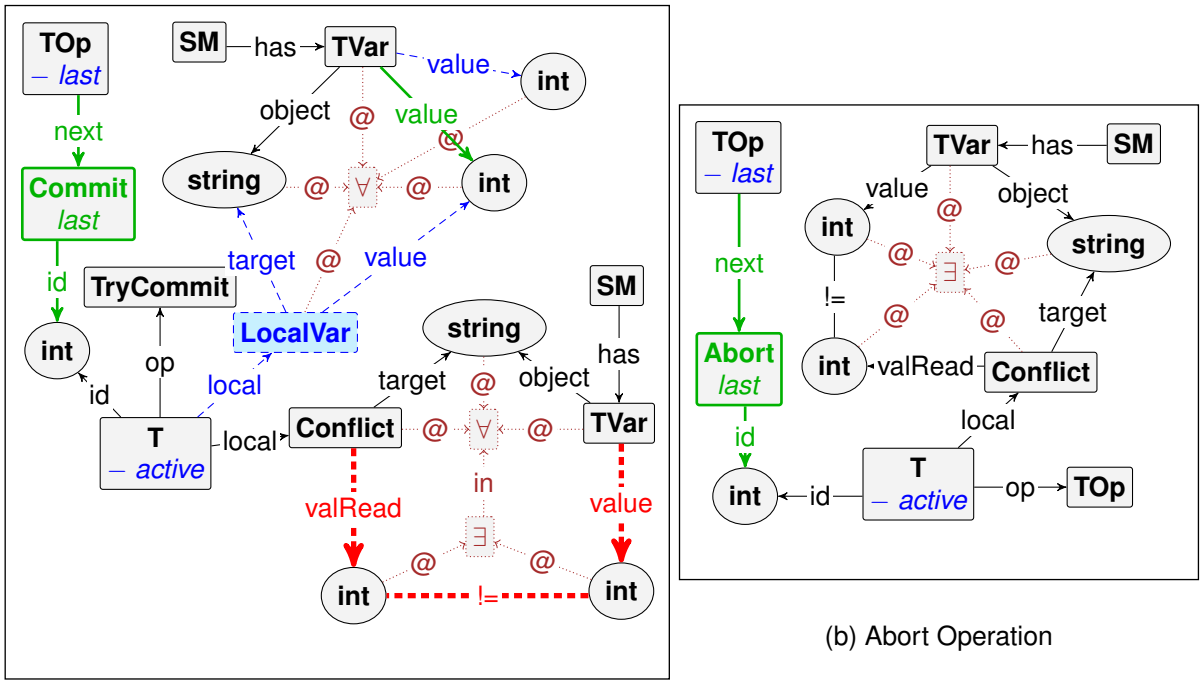
(b) Repeatable step

(c) Last step

Figure 21 – 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 and it will be reflected in the graph representation of the operations. In an algorithm with eager conflict detection, some transaction is likely to be aborted at any point if a conflict happens. This can approached by always checking the version of a variable read by the transaction. As shown in Figure 22, a *local* node *conflict* stores the value read of each variable the transaction performed a read operation on, this can be used as validation that the transaction has read a stable state of the system. Figure 22(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 (*TVar* node in the *SM*) 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 22(b), with at least one conflict the rule can be triggered by using the quantifier *exists* ($\exists$). Both instances of verification are mutually exclusive, a transaction cannot choose to commit (or read or write for that matter) and abort at the same time.



(a) Commit Operation

(b) Abort Operation

Figure 22 – Example of production rules for a Commit and Abort with lazy versioning and eager conflict detection.

In the case of lazy conflict detection, instead of having an abort operation that can be called at any point, the production rules that finish the execution of a transaction (either to commit or abort) have opposing matching conditions when a *TryCommit* is called: the production rule for a commit only matches if there are no conflicts, on the other hand, the production rule for an abort only matches if at least one conflict is found.

4.3 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 13 (in addition of the global nodes such as *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 23.

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

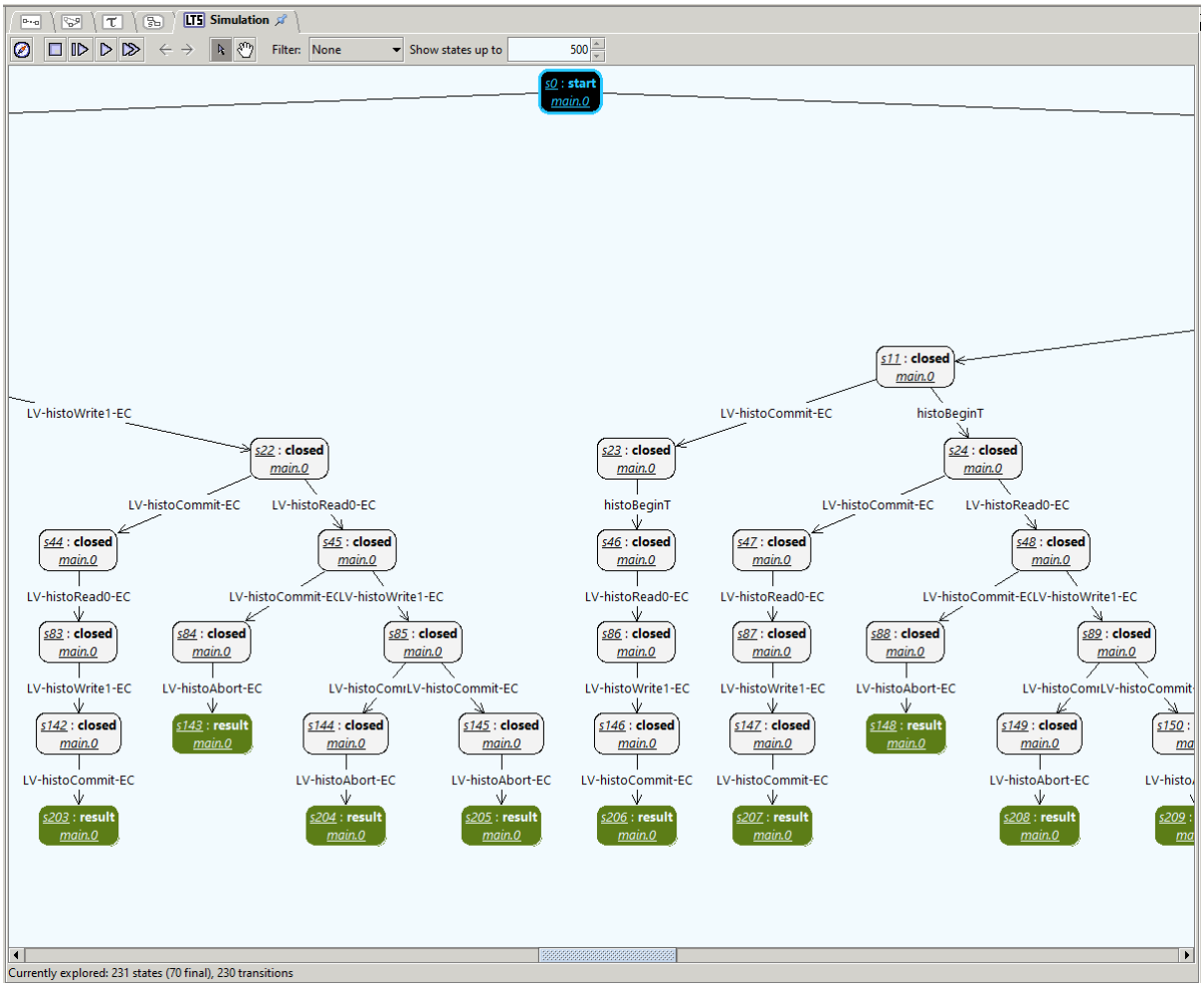


Figure 23 – Labelled Transition System Simulation in the GROOVE tool.

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, in the case for the STM system 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.

Those green final nodes are the target of the LTS in this work. The LTS demonstrated in Figure 23 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*. This will be elaborated further in this chapter.

4.4 Correctness Criteria

Having defined the basic rules that will create the state space containing all histories of a TM algorithm, the next step is to verify their correctness. This verification will be based on a graph characterization by Guerraoui; Kapałka (2010), where a conflict graph is built by analysing the operations in a given history. In this approach the conflict

graph is created at the same time as the history. This is accomplished by augmenting the production rules with the verification for conflicts.

The properties discussed in Chapter 2.2 and Chapter 2.3 are the main focus for the correctness criteria test. A history can be *sequential* if no two transactions are concurrent. It can also be *complete* if there are no live transactions, meaning that all transactions finish with an abort or commit operation. And finally, histories and transactions can be *legal*, if all transactions in a sequential history respect the sequential specifications of all shared objects.

In the graph grammar representing an algorithm, *T* is used as the main vertex for a transaction and its operations, whereas *Tran* is the vertex representing the transaction in the conflict graph. To keep a consistent nomenclature, the conditions described in in Chapter 2.3.2 to create the opaque graph OPG are applied over *Tran* vertices instead. Rewriting the definitions to fit the current context result in the following. According to Guerraoui; Kapałka (2010), to avoid dealing with the initial values of *TVars* separately from the values written to those *TVars* by transactions, a "virtual" committed initializing transaction $Tran_0$ needs to be introduced. $Tran_0$ writes value 0 to every *TVar* (in every TM history). Let *H* be any TM history. Let $V_{\ll}$ be any version order function in *H*. Denote $V_{\ll}(x)$ by $\ll_x$. An opaque graph $OPG(H, V_{\ll})$ is a directed, labelled graph constructed by following the rules:

1. For every transaction $Tran_i$ in H (including $Tran_0$) there is a vertex $Tran_i$ in graph $OPG(H, V_{\ll})$. Vertex $Tran_i$ is labelled as follows: vis if $Tran_i$ is committed in H or if some transaction performs a read operation on a *TVar* written by $Tran_i$ in H, and loc, otherwise.

2. For all vertices $Tran_i$ and $Tran_k$ in graph $OPG(H, V_{\ll})$, $i \neq k$, there is an edge from $Tran_i$ to $Tran_k$ (denoted $Tran_i \rightarrow Tran_k$) in any of the following cases:

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

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

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

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

Consistency in a TM means that every time a transaction $Tran_i$ reads a *TVar* x, the value returned is either the latest value written by $Tran_i$ in x, or any value written by another committed or commit-pending transaction. In (GUERRAOUI; KAPAŁKA, 2010), the authors define that a consistent TM history H that has unique writes is opaque (GUERRAOUI; KAPALKA, 2008) if, and only if, there exists a version order function $V_{\ll}$ in H such that graph $OPG(H, V_{\ll})$ is acyclic. The proof can be found in (GUERRAOUI; KAPAŁKA, 2010). Figure 24 shows an example of an opaque history, the corresponding conflict graph can be seen in Figure 25.

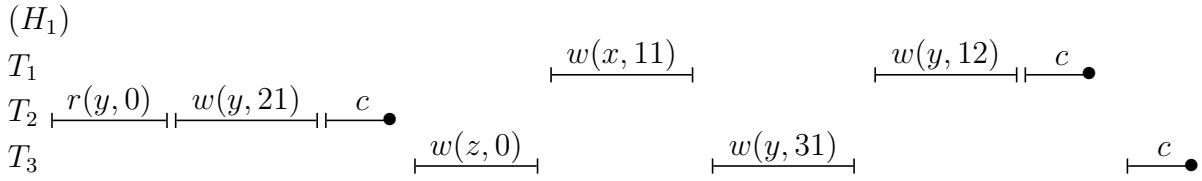


Figure 24 – Example of opaque history.

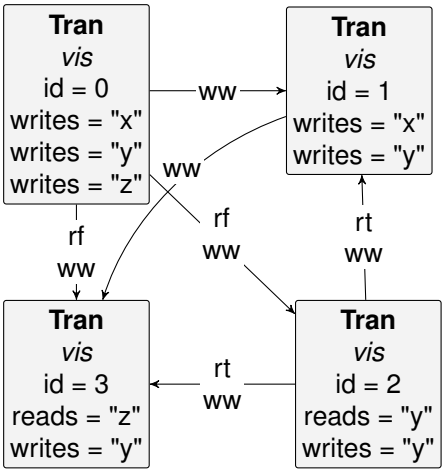
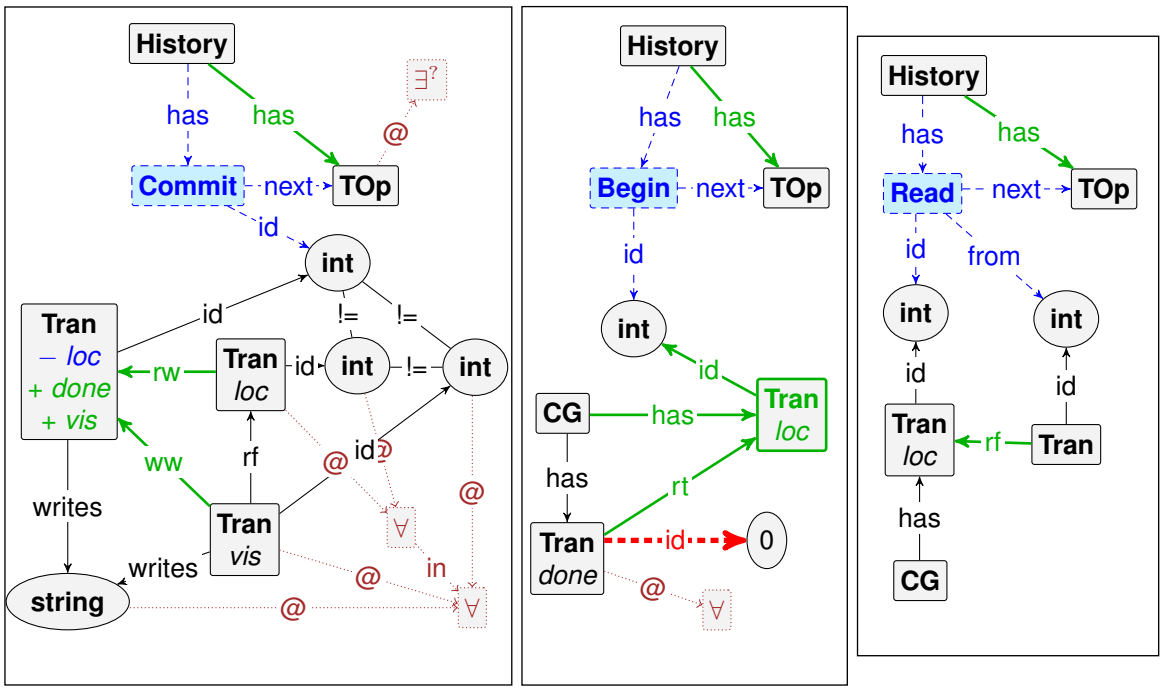


Figure 25 – Conflict graph for an opaque history.

The proposed methodology to formalize STM algorithms into graph grammars is able to analyse the entire LTS constructed above and verify correctness of each history. It was observed that 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 *Tran*-nodes, that represent each transaction, which results in very simple production rules.

Figure 26 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 *T* with an *op*-edge was used to identify the current operation of the various parallel transactions, but now because a history is evaluated individually, the node *History* itself is used to path through the sequence. The *has*-edge on the node *History* always starts by pointing to the first element of the history. The operations in Figure 26 cover all four conflicts defined by Guerraoui; Kapałka (2010): Figure 26(a) shows a commit operation

that creates read before write (*rw*) and write before write (*ww*) relations; Figure 26(b) shows a begin operation that creates real-time (*rt*) relations. Figure 26(c) shows a read operation that creates reads from (*rf*) relations.



(a) Commit Operation (b) Begin Operation (c) Read Operation

Figure 26 – Example of production rules for the Conflict Graph *Commit*, *Begin* and *Read* operation.

4.5 Computation Tree Logic

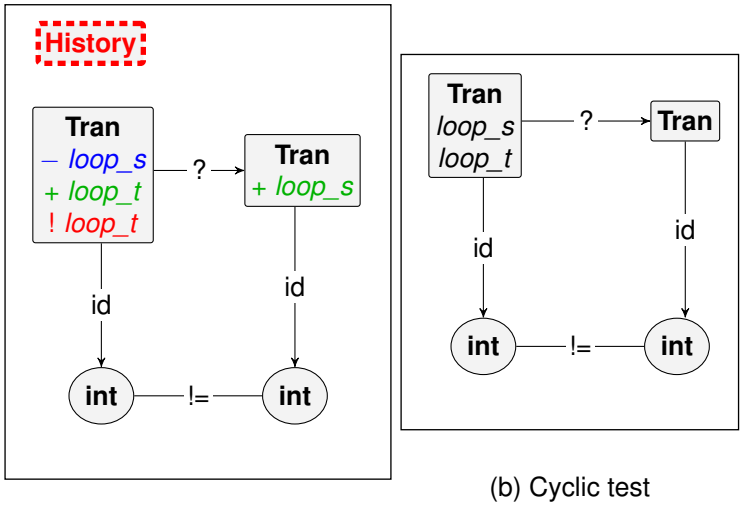
Computation Tree Logic (CTL) is a branching-time logic that has proved to be a useful and versatile framework for reasoning about properties of distributed systems (EMERSON; CLARKE, 1982; EMERSON; HALPERN, 1986). Temporal logics, such as CTL, offer facilities for the specification of properties that the behavior of the system must fulfill (CLARKE; EMERSON; SISTLA, 1986).

The CTL verification tool on GROOVE is used to check if any final state of the evaluation has a cyclic conflict graph or not, this is achieved by using a *graph condition* rule: a rule that does not create or delete objects. At the end of the history generation some auxiliary production rules are used to path through the conflict graph and mark each node it passes, creating a spanning tree by following directed edges starting on $Tran_0$. The graph condition *cyclic* simply looks for a match on a path that crossed an already marked node, forming a cycle.

The verification is made by the CTL formula *AG !cyclic*. This formula states that for every path following the current state, the entire path holds the property of not pattern

matching the graph condition *cyclic*. Figure 27 shows the production rule that marks the CG looking for cycles and the graph condition used in the CTL. In this figure a feature of GROOVE is used, called “wildcard”, in the form of a “?” edge connecting two vertices. This edge means that any possible labelled edge can be used in the patter match for the rule, in this case they are *rf*, *rt*, *rw* and *ww*.

The entire process of analysing the correctness of an STM algorithm using graph grammar is composed of: first, generating histories; then a conflict graph is extracted for each history, overlaps may occur where multiple histories result in the same CG; then each CG is marked via the production rule in Figure 27(a); finally, the CTL is formula is executed using the production rule in Figure 27(b). If no matches occur, no conflict graphs had cycles.



(a) Loop pathing in the CG

(b) Cyclic test

Figure 27 – Production rules for the test of acyclicity in a conflict graph.

4.6 Final Remarks

This chapter described the methodology to verifying opacity as a correctness criterion for Transactional Memory (TM) algorithms via a translation to a Graph Grammar (GG). The graph characterization of opacity is well known in the literature and it was used to demonstrate opacity of every history the algorithm generates for a given entry program. A graph representation of sequential operations guides how the system is transformed in the approach described in this chapter, composing concurrent transactions of the initial state and the series of operations in histories. Production rules were used to “execute” a full transactional operation in a single step, transforming the state of the system accordingly. Because the initial state includes more than one transaction, the non-determinism of which transaction operation to execute creates multiple orders of operations. The labelled transition system generated by the tool GROOVE

represents all possible sequences of rule applications, and at each step, it adds to the history which operation was executed. Any leaf of this tree contains a full history, thus the set of all histories is equal to the set of all possible rule application orders. It is worth noting that this set of histories is directly dependent on the set of transactions given in the initial state.

Given that a set of histories can be generated from a TM algorithm, evaluating their correctness can be equivalent to evaluating the correctness of the algorithm. The graph characterization of opacity was used for this purpose. When translating the algorithm into a graph grammar, extra steps can be added after the histories are generated to apply the opacity verification. These steps create a conflict graph for each history and apply a logic of flagging nodes to look for loops in the conflict graph. In the same sense that an input of transactions results in non-determinism for the order of execution of operations, adding flags to the conflict graph also results in different paths being followed. If any of those paths results in a cycle means that at least one of the reachable states will match on the acyclicity test. In the tool GROOVE, this was done using computation tree logic. The first iteration of graph grammars as a method to extract conflict graph of a single history resulted in the peer reviewed paper seen in (CARDOSO; FOSS; BOIS, 2019). As for the methodology described in this chapter, that applies the concept of this first iteration for an entire set of histories, resulted in the peer reviewed paper seen in (CARDOSO; FOSS; BOIS, 2021). An alternative to using the tool GROOVE to apply this correctness test is the event-B model that will be described in the next chapter. The main reason to present an alternative solution to this step of the formalism is the fact that GROOVE needs to create the entire state space based on the production rules before the acyclicity test can be applied. For this reason, event-B can be a useful alternative that uses proof by induction on reachable states of the system and does not create the full state space to evaluate cycles in the conflict graphs.

5 EVENT-B ALTERNATIVE FOR CORRECTNESS VERIFICATION

The vast majority of existing approaches for verifying graph transformation systems follows the model checking paradigm. It explores, which configurations are reached when transformation rules are applied to a given start graph. Typical properties under investigation are therefore whether particular invariants are maintained during graph rewriting, or whether certain configurations are reachable. Model checking is attractive because it offers a high degree of automation and is therefore accessible also to uninitiated users. The work of (DA COSTA; RIBEIRO, 2009, 2012) presents a logical model for reasoning about graph transformations. This approach has been implemented in Event-B (RIBEIRO et al., 2010), a state-based formalism, by coding individual rules as event-B machine events and then profiting from the proof support for this platform to prove the correctness of rules. Further work has also been made to introduce negative application conditions and attributes (COSTA CAVALHEIRO; FOSS; RIBEIRO, 2017).

The methodology to formalize Transactional Memory (TM) algorithms using graph grammar includes a step of generating a state space that contains different order of executions of operations to the shared memory, and a step of applying a correctness test to these executions. The result of this test is the extraction of a conflict graph and the verification of its cycles. If there are no cycles in the conflict graph, the correctness criteria used deems the execution as correct.

In the formalization described in Chapter 4 and the study cases presented in Chapter 6, the tool GROOVE is used to create the state space via a Labelled Transition System (LTS), extract the conflict graph and use the Computation Tree Logic (CTL) feature to check for cycles. However, because GROOVE is a primarily visual tool, the state space created includes the visual data for each reachable state. This can result in unnecessary processing power being used for these graphs when in reality the goal is to check for a pattern match (graph condition) in all of them without actually needing to access them individually. Another point that can be made about using GROOVE's state space generator is that depending on the size of the starting graph, the non-determinism nature of TM executions can generate more and more variations

of execution orders, which only increase the size of the state space, making the optimization of this step of the methodology very important.

This chapter presents an alternative to using the CTL feature of GROOVE to check for cycles in conflict graphs by modelling the evaluation of a history in Event-B. This does not include the entire state space generation as that would require the TM algorithm to also be modelled in Event-B, that is envisioned for future works. The graph grammar presented in this chapter only contains a history, a sequential set of operations to the shared memory, and the transformation that production rules apply is used to create the conflict graph and analyse its acyclicity. Section 5.1 describes the necessary definitions of an even-B model. Section 5.2 presents the modified graph grammar used in the correctness verification in this chapter. Section 5.3 presents the event-B translation of the new GG and the verification process developed.

5.1 Event-B Modeling

Event-B (ABRIAL; HALLERSTEDE, 2007; ABRIAL, 2010) is a state-based formalism that is related to Classical B (ABRIAL; HOARE, 1996) and Action Systems (BACK; SERE, 1989). This section presents the definition of this formalism, symbols and operations presented that will be used in the remainder of this text. Table 1 shows these symbols and their meanings, corresponding to the standard event-B mathematical notation.

Table 1 – Definition of symbols and operations (Source: (COSTA CAVALHEIRO; FOSS; RIBEIRO, 2017)).

Symbol/operation	Meaning
$\nrightarrow$	Partial functions or morphisms
$\rightarrow$	Total functions or morphisms
$\rightarrowtail$	Partial and injective functions or morphisms
$\rightarrowtail$	Total and injective functions or morphisms
$f \circ g$	Composition of functions or morphisms f and g
f^{-1}	Inverse function
$\mapsto$	Mapping relation
$\uplus$	Disjoint union
$\mathbb{N}$	Set of natural numbers
$\mathbb{P}$	Set of all subsets (power set)
$\setminus$ or $-$	Set difference
rng(r)	Range of a binary relation r
dom(r)	Domain of a binary relation r
card(A)	Number of elements of set A
$r[A]$	$\{y \mid \exists x \cdot x \in A$ and $x \mapsto y \in A\}$ (relational image)
$\lhd$	$A \lhd r \stackrel{\text{def}}{=} \{(a,b) \in r \mid a \notin A\}$ (domain subtraction)
$\lhd\!\!\!-$	$r \lhd\!\!\!- s \stackrel{\text{def}}{=} (dom(s) \lhd r) \cup s$ (relation overriding)
$\rhd$	$r \rhd B \stackrel{\text{def}}{=} \{(a,b) \in r \mid b \in B\}$ (range restriction)

An event-B model consists of two parts: a static part called context and a dynamic part called machine. Event-B is based on First-Order Logic with Set Theory.

Definition 19 (Event-B model, event)**.** *An **event-B model*** E B Model = $(\mathcal{C}, \mathcal{M})$ *is defined by a **context*** $\mathcal{C}$ = (c, s, A) *and a machine* $\mathcal{M}$ = (v, I, init, E)*, where* c *and* s *are sets of constant and set names, respectively;* A(c, s) *is a collection of axioms constraining* c *and* s*;* v *is a set of state variables;* I(v) *is a model invariant limiting the possible assignments to* v, $\exists$ c, s, v · A(c, s) $\wedge$ I(v) - *i.e.* A *and* I *characterize a non-empty set of model states;* init(v') *is an initialization action assigning initial values to the model variables; and* E *is a set of model events. An event is a tuple* e = (G, BA) *where* G(v) *is a formula, called the guard, and* BA(v, v') *is a before-after predicate Types of constants must be defined as axioms in* A*, whereas types of variables must be defined as invariants in* I*.*

The concrete syntax of the event-B Camille editor (from the Rodin platform), as shown in Figure 28, will be used as representation of the event-B model. In the context, sets and constant names are defined, and arbitrary axioms may be used, the only requirement is that types of constants must be defined as axioms. In the machine, variables can be declared and must have a type defined by an invariant. Invariants are also used to describe the desired properties of reachable states of a system.

Figure 28 – Event-B syntax example (Camille editor).

Besides variables and invariants, events are also included in the machine, they have the ability to transform the state of the system. The *initialization* event does not

have guards (requirements to trigger its execution), and must assign a value for each variable of the machine. These assignments may be non-deterministic, but this feature is not used in this work since a concrete initial graph is used in the graph grammar. To define the other events that describe the behavior of the system, auxiliary variables can be used (in the *any* block), that must be typed in the guards of the event (*where* block). The guards are also used to specify the conditions that must hold in a state for the event to be enabled. Finally, the *then* block implements the before-after predicate: it is used to assign the new values to variables. Not all variables must be changed in an event, the values of the ones that are not listed remain unchanged.

Model correctness is demonstrated by generating and discharging a collection of proof obligations that ensure that the initial state is feasible and satisfies all invariants of the model. If any event can be executed at a state that satisfies all invariants, it will result in a state that also satisfies all invariants.

Definition 20 (Model correctness)**.** *Given an event-B model* E B Model = ($\mathcal{C}$, $\mathcal{M}$)*, with* $\mathcal{M}$ = (v, I, init, E)*, and an event* ev = (G,BA) $\in$ E *or* ev = init*. The event* ev *is correct if the following conditions are satisfied:*

- *Feasibility (FIS):* $\begin{cases} I(v) \wedge G(v) \implies \exists v' \cdot BA(v, v'), & \text{if } ev \in E \\ \exists v' \cdot init(v'), & \text{if } ev = init \end{cases}$
- *Invariant Satisfaction (INV):* $\begin{cases} I(v) \wedge G(v) \wedge BA(v, v') \implies I(v'), & \text{if } ev \in E \\ I(v'), & \text{if } ev = init \end{cases}$

An event-B model is correct if all its events are correct.

The feasibility condition (FIS) states that whenever the invariants and the guard of an event are true in some state *v*, it is possible to obtain a state v' that satisfies the before-after predicate of this event. To ensure the feasibility of the system, all sets in a context are assumed to be non-empty in an event-B model (sets represent types, and thus empty types are not allowed). The invariant satisfaction condition (INV) ensures that if the event occurs, it brings the system to a state v' that satisfies all model invariants. Properties that a model should exhibit are described as invariants, and thus proving that a model is correct means proving that each event does not bring a system to a state in which the desired properties do not hold. Given a model that is correct, the behavior of an event-B model is defined by a transition system, as described below.

Definition 21 (Event-B model behavior)**.** *Given a correct model* E B Model = ($\mathcal{C}$, $\mathcal{M}$)*, with* $\mathcal{C}$ = (c, s, A) *and* $\mathcal{M}$ = (v, I, init, E)*, its behavior is given by a transition system* BST = (BState, BS_0, $\rightarrow$) *where:* BState = {$\langle v \rangle$} | v are state variables and $\langle v \rangle$ is a valuation of v} $\cup$ {Undef}*,* BS_0 = Undef*, and* $\rightarrow \subseteq$ BState $\times$ BState *is the transition relation given*

by the rules:

$$\textit{(start)}\frac{\textit{init}(v')}{\textit{Undef} \rightarrow \langle v' \rangle} \qquad \textit{(transition)}\frac{\exists(\textit{G, BA}) \in \textit{E} \cdot \textit{I}(v) \wedge \textit{G}(v) \wedge \textit{BA}(v, v')}{\langle v \rangle \rightarrow \langle v' \rangle}$$

The model is initialized (rule *start*) to a state described by *init*(v') and then, as long as there is an enabled event (rule *transition*), the model may evolve by firing this event and computing the next state according to the event's before-after predicate. Events are atomic. In case more than one event is enabled at a certain state, demonic choice semantics applies (BACKHOUSE; WOUDE, 1993; BERGHAMMER; SCHMIDT, 1993). Note that model correctness implies that all reachable states satisfy the invariants of the model.

In an event-B model, refinement is a fundamental concept since the idea is to construct a model for a system in steps. The construction starts with a very abstract model which is refined until it includes requirements and is at the desired level of abstraction. The refinement of an abstract model $\mathcal{M}^A$ is a concrete model $\mathcal{M}^C$ that is behaviorally related to the abstract one. In event-B, relating the models is achieved by constructing a refinement mapping between $\mathcal{M}^C$ and $\mathcal{M}^A$ and by discharging a number of refinement proof obligations. These proof obligations ensure that all computations that are possible at the concrete level were also possible at the abstract level, meaning that no new behavior with respect to the variables that compose the abstract state was introduced in the concrete model. When a concrete model is constructed as a refinement of an abstract model, it is not necessary to redo any proof to guarantee that the concrete model satisfies the properties stated as invariants of the abstract model.

A refinement $\mathcal{C}^C$ of a context $\mathcal{C}^A$ is obtained by adding new sets, constants and axioms. A concrete machine, which sees a concrete context $\mathcal{C}^C$, can use all concrete sets and constants, as well as the abstract ones (in $\mathcal{C}^A$). In a machine refinement, the set of variables of a concrete machine $\mathcal{M}^C$ must be completely disjoint from the collection variables of the corresponding abstract machine $\mathcal{M}^A$. In this work, refinement was not used because the level of abstraction needed did not require it, however, for further improvements on the approach described here, it is expected to be necessary a resource.

An extensive tool support through the Rodin Platform makes event-B especially attractive (DEPLOY; RODIN, 2023). An integrated Eclipse-based development environment is actively developed, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is theorem proving supported by a collection of theorem provers, but there is also some support for model checking.

5.2 GG for Transactional Memory

The type graph represents the correct structure each state must adhere to in the system, an example can be seen in Figure 29. In this notation a supertype *TOp* (Transactional Operation) is used to represent the multiple operation nodes in one. The node *History* points to the current operation, that can be in a normal state (*OP*), or in a lock/release state. If the node *History* points to *End*, it will be processed as the end of the execution. A *CGNode* has flags to denote its state: done, visible or local. If a transaction commits or aborts, it will be considered done. Moreover only committed transactions are flagged as visible, they are local otherwise. The *CGNode* stores which variables the transaction wrote to, and which relations to other nodes it has (*RF*, *RT*, *RW* and *WW*). Lastly, the node *CG* is used to lock or free *CGNodes* when committing or starting transactions.

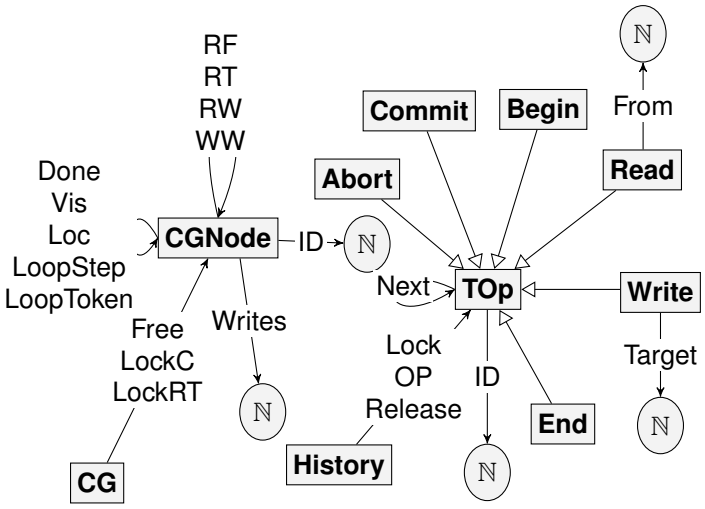


Figure 29 – Type graph.

The initial state includes a full history and an empty conflict graph. Processing each operation will populate the conflict graph to be evaluated later. An operation node can have attributes and a *next*-edge pointing to the next operation to be processed. At the end of the history a node *End* is used to tell the system the history processing is over, and the conflict graph can be looked at. Figure 30 shows and example of initial state. From this graph onwards, nodes, edges and attributes have a label attached to them, this is used in the event-B translation later.

5.2.1 Transactional operations

A **begin operation** is divided in 5 total production rules, each one represents an internal step the operation needs. These steps represent a combination of: locking the system, to only allow this operation to execute; dealing with any “loop” necessary, transforming multiple nodes one at a time; and any unlocking needed to transition between steps. The reason this operation had to be divided in 5 steps is mostly because

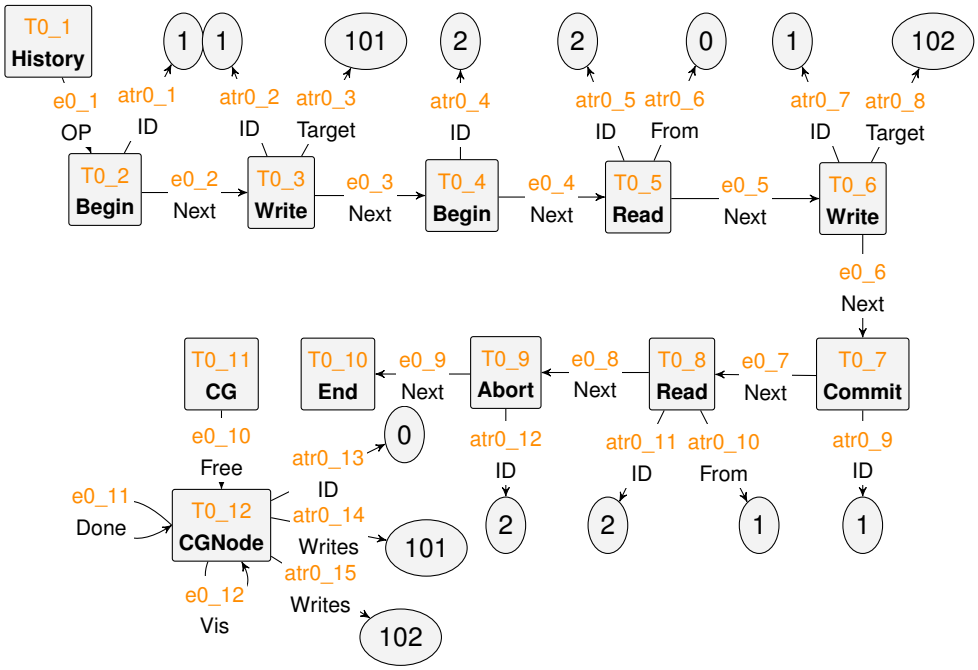


Figure 30 – Initial state graph.

a loop cannot be represented without more complex operators. The tool GROOVE can allow this notion with the use of quantifiers “for all” and “exists” (∀ and ∃), which makes all 5 steps be executed with one production rule. To save some space, Figure 31 shows the GROOVE version of the begin operation, but in Appendix B all 5 production rules can be seen.

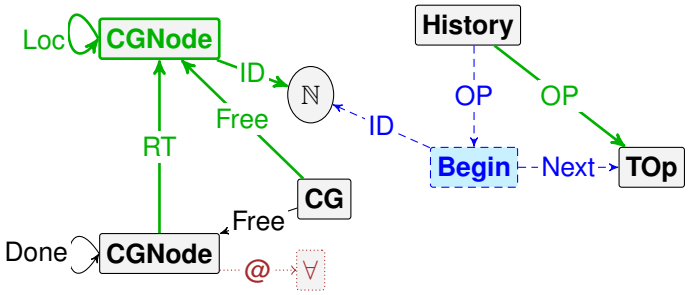


Figure 31 – *Begin* operation in a single step.

The 5 steps of a *begin* operation are divided in the following production rules:

- **BeginLock rule (Figure 56):** Locks the history into a *begin* operation (nothing else can execute), and creates a *CGNode* for the corresponding new transaction.
- **BeginLoop1 rule (Figure 57):** Loops through any **free** and **finished** transaction that does not have a real-time relation (*RT*-edge) with the new transaction and creates the new edge signifying the relation.
- **BeginLoop1_Release rule (Figure 58):** When there are no more transactions to

mark with real-time relations, marks the history for the second loop. This prepares to release of the execution for other transactions.

- **BeginLoop2 rule (Figure 59):** Loops through any locked *CGNodes* that had a new edge added, and frees them.
- **BeginLoop2_Release rule (Figure 60):** When there are no more *CGNodes* to free (or none to begin with), releases the lock on the history and move the *OP*-edge to resume the execution of other operations.

The **read operation** creates a *reads-from* edge (*RF*) from the *CGNode* of the transaction that wrote the value to the one executing the operation. The operation can be done in one production rule, so it only takes one step. Figure 32 shows the production rule that processes a read operation.

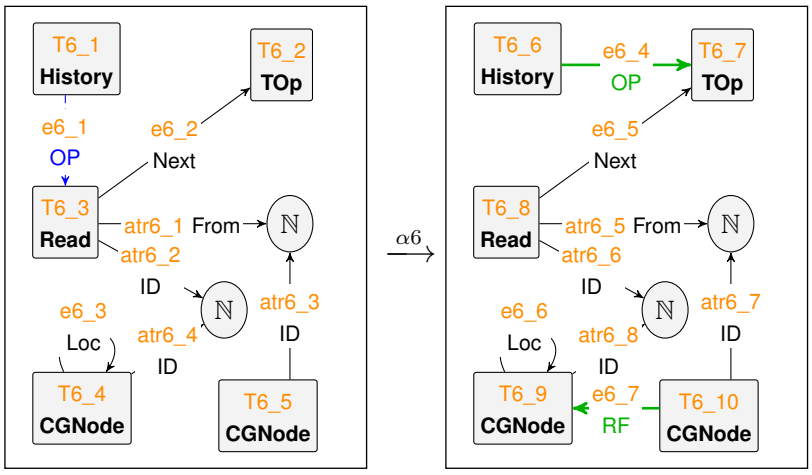


Figure 32 – *Read* production rule.

The **write operation** creates a *Writes*-edge in the *CGNode* of the transaction that executed the operation. This edge can be used as a flag for *write-before-write* conflicts when a transaction commits. The operation can be done in one production rule, so it only takes one step. Figure 33 shows the production rule that processes a write operation.

The **commit operation** is the one that finalizes the execution of a single transaction, making its changes visible to other transactions. Because of the nature of the changes it has to make, it needs to analyze multiple nodes of the conflict graph. The solution for this is to use a loop, in a similar way as the *begin* operation, so for the full operation to execute 5 steps are needed. The graph in Figure 34 shows the full operation using the *for all* ($\forall$) quantifier, this graph represents the full operation. The two nested quantifiers used here can be read as: for all *CGNodes* that write to the same variable as the main transaction, a *WW*-edge is created; at the same time, for all *CGNodes* that read any

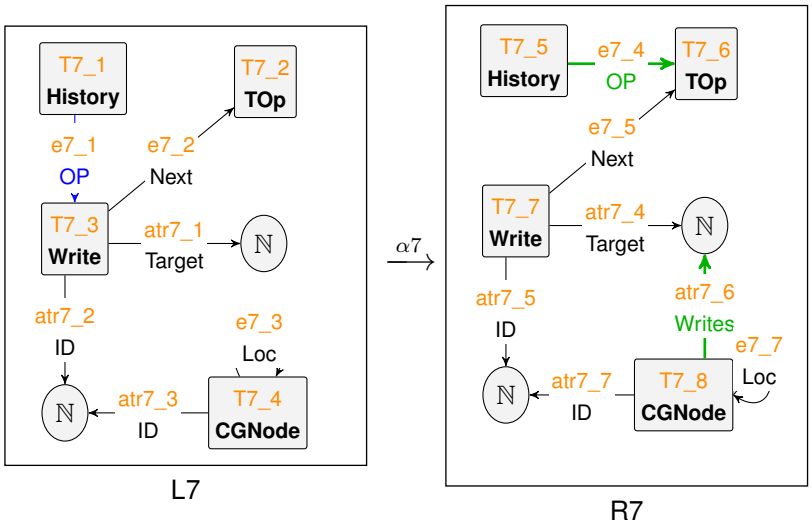


Figure 33 – *Write* production rule.

of these variables (that were overwritten by the main transaction), a new *RW*-edge is created.

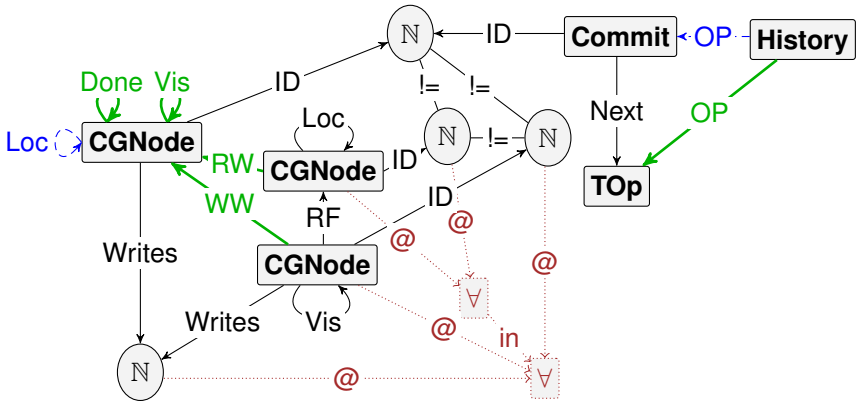


Figure 34 – *Commit* operation in a single step.

The logic behind a commit operation divide in 5 steps is as follows:

- **CommitLock rule:** Locks the history into a *commit* operation, no other operation can execute.
- **CommitLoop1 rule:** Loops through visible transactions that wrote to the same variables as the transaction executing the commit. Add the new edge *WW*, denoting *write-before-write* relation between them.
- **CommitLoop1_Release rule:** As soon as there are no more *WW*-relations to be added, prepare the operation for the second loop.
- **CommitLoop2 rule:** Loop through any transaction that read a value overwritten by the main trasaction (has a *WW*-relation). Adds the *RW*-edge between

the transaction that read the variable and the main one. This denotes the *read-before-write* relation, where a value was read and overwritten afterwards, making it inconsistent.

- **CommitLoop2_Release rule:** Lastly, if there are no more *RW*-edges to add, finishes the execution and unlocks the history for other operations to execute.

The **abort operation** terminates the execution of a transaction, flagging it as "done". However, because it aborted, its changes are not visible to other transactions. It is important to flag even aborted transactions because the *begin* operation creates the real-time relation with these as well. Figure 35 shows the production rule responsible for an *abort* operation, where the *OP*-edge is moved along to the next one in line, and a new edge is added to the *CGNode* to flag it as "done".

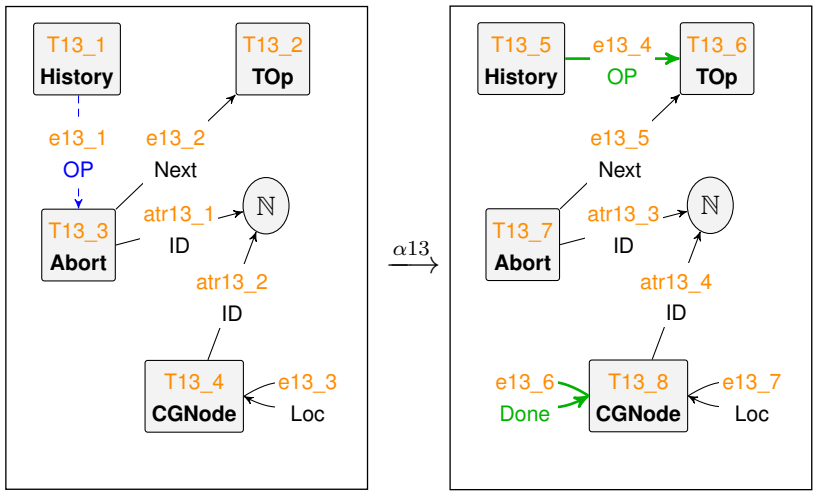


Figure 35 – *Abort* production rule.

5.2.2 LoopStart and LoopStep rules

Last step of transformations the system does. The objective is to path through the generated Conflict Graph to find cycles. The main logic of the pathing is to use a marker to choose the next step (edge *loopStep*, can be seen as a flag), and leave behind a token to mark a path already visited (edge *loopToken*). The production rule that marks the CG can only be applied if a *CGNode* has a *loopStep*, but not a *loopToken*. So the idea is to path through the CG leaving behind marks, and if at any point the pathing encounters a mark that was left behind, this signifies a loop in the CG. The production rule *LoopStart* (Figure 36) starts this process, and *LoopStep* (Figure 37) executes the remainder of it.

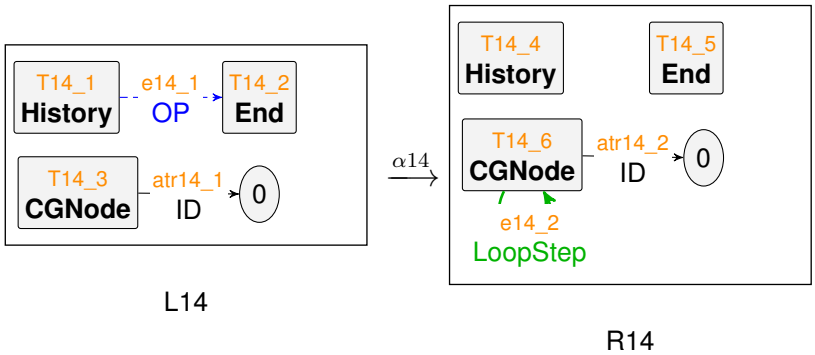


Figure 36 – *LoopStart* rule: starts the Conflict Graph pathing process.

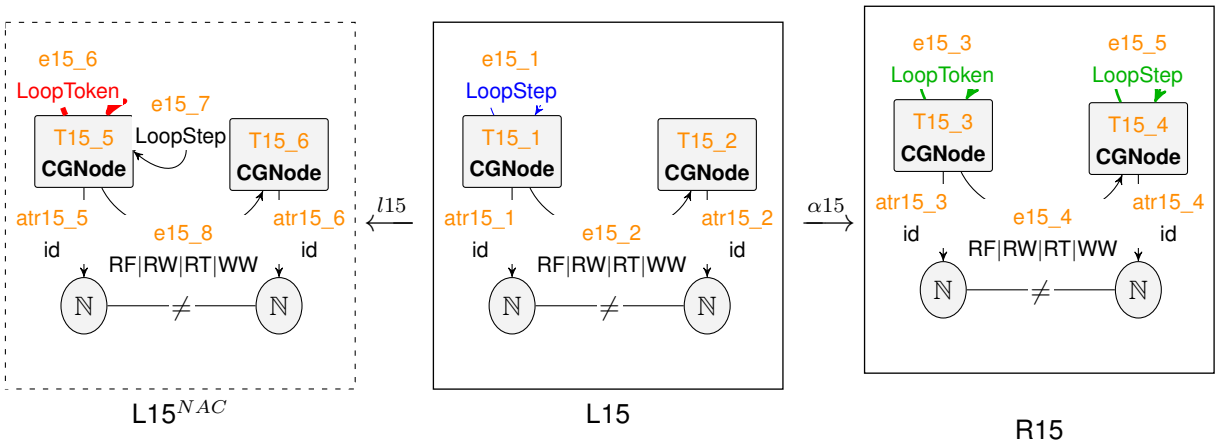


Figure 37 – *LoopStep* rule: marks the path of the Conflict Graph.

5.3 Event-B for Transactional Memory

This section presents the event-B translation of the GG described previously. An event-B model includes two parts: a context and a machine. The context includes the global type graph of the system and a type definition for each production rule. The machine part includes the initial graph and the main body of the production rules (the condition for pattern matching and changes to be made when executed).

5.3.1 GG Type Graph and Initial State

When translating the **type graph** for the GG into an event-B model, three sets are used: a set of vertices, a set of edges and a set of attributes. Vertices have edges and attributes pointing to and from them. The only difference between edges and attributes is that edges connect two vertices, meanwhile attributes connect a vertex to a natural number. The context elements below represent a type graph as seen in Figure 29. The three main sets *vertT*, *edgeT* and *attrT*, which are defined as partitions with their respective constants. To deal with edges, *sourceT* and *targetT* are functions that define where each edge can point to. In a similar way as edges, attributes use the functions *attrN* (attribute's node) and *attrV* (attribute's value) to show their mapping in the type graph. In this case all attributes use natural numbers as their value.

```
sets
  vertT
  edgeT
  attrT
constants
  CG CGNode History End Begin Read Write Commit Abort
  RF RT RW WW
  Vis Done Loc LoopStep LoopToken
  Free LockRT LockC
  OP Release Lock Next
  ID Writes Target From
axioms
  @axm_vertT: partition(vertT, {CGNode}, {CG}, {Begin}, {Read}, {Write}, {Commit}, {Abort},
                        {History}, {End})
  @axm_edgeT: partition(edgeT, {RF}, {WW}, {RW}, {RT}, {LoopStep}, {LoopToken}, {Next}, {OP},
                        {Done}, {Vis}, {Free}, {LockC}, {LockRT}, {Loc}, {Lock}, {Release})
  @axm_attrT: partition(attrT, {ID}, {Writes})
  @axm_sourceT: sourceT ∈ edgeT → vertT
  @axm_sourceT_def: partition(sourceT, {Lock ↦ History}, {Release ↦ History}, {LockC ↦ CG},
                        {LockRT ↦ CG}, {Loc ↦ CGNode}, {Free ↦ CG}, {Done ↦ CGNode},
                        {Vis ↦ CGNode}, {RF ↦ CGNode}, {WW ↦ CGNode}, {RW ↦ CGNode},
                        {RT ↦ CGNode}, {LoopStep ↦ CGNode}, {LoopToken ↦ CGNode},
                        {OP ↦ History})
  @axm_targetT: targetT ∈ edgeT → vertT
  @axm_targetT_def: partition(targetT, {Lock ↦ Begin}, {Lock ↦ Commit}, {Release ↦ Begin},
                        {Release ↦ Commit}, {LockC ↦ CGNode}, {LockRT ↦ CGNode},
                        {Loc ↦ CGNode}, {Free ↦ CGNode}, {Done ↦ CGNode}, {Vis ↦ CGNode},
                        {RF ↦ CGNode}, {WW ↦ CGNode}, {RW ↦ CGNode}, {RT ↦ CGNode},
                        {LoopStep ↦ CGNode}, {LoopToken ↦ CGNode}, {OP ↦ Begin},
                        {OP ↦ Read}, {OP ↦ Write}, {OP ↦ Commit}, {OP ↦ Abort}, {OP ↦ End})
  @axm_attrN: attrN ∈ attrT → vertT
  @axm_attrN_def: partition(attrN, {ID ↦ CGNode}, {Writes ↦ CGNode}, {Target ↦ Write}, {From ↦ Read})
  @axm_attrV: attrV ∈ attrT → ℙ(ℕ)
  @axm_attrV_def: partition(attrV, {ID ↦ ℕ}, {Writes ↦ ℕ}, {Target ↦ ℕ}, {From ↦ ℕ})
```

The **initial state** graph, as seen in Figure 30, includes a history (sequence of operations) and an empty conflict graph. The translation into an event-B model will include global variables that define the state of the system, all production rules can see and modify these variables, therefore modifying the current state of the system. In the code below, invariants are used to give a type to variables. These variables are defined as sets and functions, in a similar way as the type graph. In addition, some properties for the system can be added here, by the nature of event-B modeling, these properties are applied to every reachable state of the system. This example includes properties that evaluate the vertices, edges and attributes' sets as finite; and the property that describes the acyclicity of the conflict graph. Now, for the contents of the initial graph itself, this is defined in the event named *INITIALISATION* that instantiates the sets and functions to be used by the production rules later.

As seen in previous figures, the nodes, edges and attributes in the production rules have a unique label attached to them. This label is used in the event-B translation as each element has to be typed in the context section of the model. The nomenclature of labels follows the pattern of: a prefix to indicate which rule the element belongs to (*T1*,

e1 and *attr1* for the first production rule, *T2*, *e2* and *attr2* for the second, and so on); followed by a number counting each element (*T1_1*, *T1_2*, *T1_3* are three vertices in the first production rule). For the initial state, the label prefix is not used, however the number counting each element still serves as an identifier for the vertices, edges and attributes.

variables
  vertG
  edgeG
  attrG
  sourceG
  targetG
  attrNG
  attrVG
  tG_V
  tG_E
  tG_A
invariants
  @inv_vertG: vertG ∈ ℙ(ℕ)
  @inv_edgeG: edgeG ∈ ℙ(ℕ)
  @inv_attrG: attrG ∈ ℙ(ℕ)
  @inv_sourceG: sourceG ∈ edgeG → vertG
  @inv_targetG: targetG ∈ edgeG → vertG
  @inv_attrNG: attrNG ∈ attrG → vertG
  @inv_attrVG: attrVG ∈ attrG → ℕ
  @inv_tG_V: tG_V ∈ vertG → vertT
  @inv_tG_E: tG_E ∈ edgeG → edgeT
  @inv_tG_A: tG_A ∈ attrG → attrT
  @inv_finV: finite(vertG)
  @inv_finE: finite(edgeG)
  @inv_finA: finite(attrG)
  @inv_propAcyclic: ∀e1,e2· e1 ∈ edgeG ∧ tG_E(e1) = LoopToken ∧
                    e2 ∈ edgeG ∧ tG_E(e2) = LoopStep ⟹ sourceG(e1) ≠ sourceG(e2)
events
event INITIALISATION
  then
    @act_vertG: vertG := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
    @act_edgeG: edgeG := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
    @act_attrG: attrG := {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
    @act_attrNG: attrNG := {1 ↦ 2, 2 ↦ 3, 3 ↦ 3, 4 ↦ 4, 5 ↦ 5, 6 ↦ 5, 7 ↦ 6, 8 ↦ 6, 9 ↦ 7, 10 ↦ 8, 11 ↦ 8,
                    12 ↦ 9, 13 ↦ 12, 14 ↦ 12, 15 ↦ 12}
    @act_attrVG: attrVG := {1 ↦ 1, 2 ↦ 1, 3 ↦ 101, 4 ↦ 2, 5 ↦ 2, 6 ↦ 0, 7 ↦ 1, 8 ↦ 102, 9 ↦ 1, 10 ↦ 1,
                    11 ↦ 2, 12 ↦ 2, 13 ↦ 0, 14 ↦ 101, 15 ↦ 102}
    @act_sourceG: sourceG := {1 ↦ 1, 2 ↦ 2, 3 ↦ 3, 4 ↦ 4, 5 ↦ 5, 6 ↦ 6, 7 ↦ 7, 8 ↦ 8, 9 ↦ 9, 10 ↦ 11,
                    11 ↦ 12, 12 ↦ 12}
    @act_targetG: targetG := {1 ↦ 2, 2 ↦ 3, 3 ↦ 4, 4 ↦ 5, 5 ↦ 6, 6 ↦ 7, 7 ↦ 8, 8 ↦ 9, 9 ↦ 10, 10 ↦ 12,
                    11 ↦ 12, 12 ↦ 12}
    @act_tG_V: tG_V := {1 ↦ History, 2 ↦ Begin, 3 ↦ Write, 4 ↦ Begin, 5 ↦ Read, 6 ↦ Write, 7 ↦ Commit,
                    8 ↦ Read, 9 ↦ Abort, 10 ↦ End, 11 ↦ CG, 12 ↦ CGNode}
    @act_tG_E: tG_E := {1 ↦ OP, 2 ↦ Next, 3 ↦ Next, 4 ↦ Next, 5 ↦ Next, 6 ↦ Next, 7 ↦ Next, 8 ↦ Next,
                    9 ↦ Next, 10 ↦ Free, 11 ↦ Done, 12 ↦ Vis}
    @act_tG_A: tG_A := {1 ↦ ID, 2 ↦ ID, 3 ↦ Target, 4 ↦ ID, 5 ↦ ID, 6 ↦ From, 7 ↦ ID, 8 ↦ Target, 9 ↦ ID,
                    10 ↦ From, 11 ↦ ID, 12 ↦ ID, 13 ↦ ID, 14 ↦ Writes, 15 ↦ Writes}
  end

5.3.2 Begin Operation

Any given production rules translation to event-B is composed of two parts: the context's typing as axioms and constants; and the machine's event as guards and acts. The following code shows the context side of the first production rule that the system executes, a *BeginLock*. As seen before, the **begin operation** is divided in five steps, this is the first. The goal of these definitions in the context is to define all the sets used by the pattern match when executing the events later. The axioms here describe exactly the state seen in the L1 graph of Figure 56, as this is the pattern match for this production rule.

```
sets
  vertL1
  edgeL1
  attrL1
constants
  T1_1 T1_2 T1_3
  e1_1
  atr1_1
  sourceL1
  targetL1
  attrNL1
  attrVL1
  tL1_V
  tL1_E
  tL1_A
axioms
  @axm_vertL1 partition(vertL1, {T1_1}, {T1_2}, {T1_3})
  @axm_edgeL1 partition(edgeL1, {e1_1})
  @axm_attrL1 partition(attrL1, {atr1_1})
  @axm_sourceL1_type sourceL1 ∈ edgeL1 → vertL1
  @axm_sourceL1_def partition(sourceL1, {e1_1 ↦ T1_2})
  @axm_targetL1_type targetL1 ∈ edgeL1 → vertL1
  @axm_targetL1_def partition(targetL1, {e1_1 ↦ T1_1})
  @axm_attrNL1_type attrNL1 ∈ attrL1 → vertL1
  @axm_attrNL1_def partition(attrNL1, {atr1_1 ↦ T1_1})
  @axm_attrVL1_type attrVL1 ∈ attrL1 → ℙ(ℕ)
  @axm_attrVL1_def partition(attrVL1, {atr1_1 ↦ ℕ})
  @axm_tL1_V tL1_V ∈ vertL1 → vertT
  @axm_tL1_V_def partition(tL1_V, {T1_1 ↦ Begin}, {T1_2 ↦ History}, {T1_3 ↦ CG})
  @axm_tL1_E tL1_E ∈ edgeL1 → edgeT
  @axm_tL1_E_def partition(tL1_E, {e1_1 ↦ OP})
  @axm_tL1_A tL1_A ∈ attrL1 → attrT
  @axm_tL1_A_def partition(tL1_A, {atr1_1 ↦ ID})
```

The changes to the state are defined in the machine, as an event. The code below demonstrates how the same production rule *BeginLock* would look as an event that deletes and creates elements. In the case of a *BeginLock*, the consequences of its execution is to delete the *OP*-edge (*e1_1*), creating a new *Lock*-edge (*e1_2*) in its place. Furthermore, to create a new *CGNode*-vertex (*T1_7*) with its *Free*-edge (e1_3) and *ID*-attribute (*attr1_3*). The deleted edge is added to the *delE* set, and any new elements have an identifier represented by a natural number that is not already in each

respective set from the initial state (*vertG*, *edgeG* and *attrG*). They are also different from each other. All these definitions can be seen in the guards of the event *BeginLock*. Besides the definitions used to change the state of the system, some extra guards define the compatibility for the correct pattern match of the system (vertices, edges, attributes and their respective auxiliary functions). The actions that events take are defined after the guards, in this case all the sets defined in the initial state receive some level of changes (deletion or insertion). Vertices, edges and attributes are deleted or inserted from *vertG*, *edgeG* and *attrG*, respectively. Adding or deleting any element, also adds or deletes their type in their respective sets (*tG_V*, *tG_E* and *tG_A*). If any changes occur to *edgeG*, the same level of changes have to be made to the auxiliary functions *sourceG* and *targetG*. In a similar way, if any changes occur to *attrG*, the same level of changes occur to *attrNG* (attribute nodes) and *attrV* (attribute values).

event BeginLock
any
  mV
  mE
  mA
  T1_7
  e1_2
  e1_3
  e1_4
  atr1_3
  delE
where
  @grd_mV mV ∈ vertL1 → vertG
  @grd_mE mE ∈ edgeL1 → edgeG
  @grd_mA mA ∈ attrL1 → attrG
  @grd_delE delE = [{e1_1}]
  @grd_newT1_7 T1_7 ∈ ℕ\vertG
  @grd_newe1_2 e1_2 ∈ ℕ\edgeG
  @grd_newe1_3 e1_3 ∈ ℕ\edgeG
  @grd_newe1_4 e1_4 ∈ ℕ\edgeG
  @grd_newatr1_3 atr1_3 ∈ ℕ\attrG
  @grd_e1_2_e1_3 e1_2 ≠ e1_3
  @grd_e1_2_e1_4 e1_2 ≠ e1_4
  @grd_e1_3_e1_4 e1_3 ≠ e1_4
  @grd_vertices ∀v · v ∈ vertL1 ⇒ tL1_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL1 ⇒ tL1_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL1 ⇒ tL1_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL1 ⇒ mV(sourceL1(e)) = sourceG(mE(e)) ∧ mV(targetL1(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL1 ⇒ mV(attrNL1(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
  @act_vertG vertG := vertG ∪ {T1_7}
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e1_2, e1_3, e1_4}
  @act_attrG attrG := attrG ∪ {atr1_3}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e1_2 ↦ mV(T1_2), e1_3 ↦ mV(T1_3), e1_4 ↦ T1_7}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e1_2 ↦ mV(T1_1), e1_3 ↦ T1_7, e1_4 ↦ T1_7}
  @act_attrNG attrNG := attrNG ∪ {atr1_3 ↦ T1_7}
  @act_attrVG attrVG := attrVG ∪ {atr1_3 ↦ attrVG(mA(atr1_1))}
  @act_tG_V tG_V := tG_V ∪ {T1_7 ↦ CGNode}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e1_2 ↦ Lock, e1_3 ↦ Free, e1_4 ↦ Loc}
  @act_tG_A tG_A := tG_A ∪ {atr1_3 ↦ ID}

```
end
```

The second step of a *begin* operation is to loop through free and finished transactions in the conflict node and add a real-time relation to the transaction being created. This production rule in this step is called *BeginLoop1* (there are two possible loops in a *begin* operation), and can be seen in Figure 57. The same principles of deleting and creating edges by modifying the *edgesG*, *sourceG* and *targetG* sets happens here. However a new condition is added to the guards of the event to act as a negative application condition (NAC). In this case, the NAC is the *RT*-edge between the free and finished node and the main transaction, to add the *RT*-edge there must not be already a real-time relation between the transactions.

```
event BeginLoop1
any
  mV
  mE
  mA
  e2_8
  e2_7
  delE
where
  @grd_mV mV ∈ vertL2 → vertG
  @grd_mE mE ∈ edgeL2 → edgeG
  @grd_mA mA ∈ attrL2 → attrG
  @grd_delE delE = [{e2_3}]
  @grd_newe2_8 e2_8 ∈ ℕ\edgeG
  @grd_newe2_7 e2_7 ∈ ℕ\edgeG
  @grd_e2_8_e2_7 e2_8 ≠ e2_7
  @grd_vertices ∀v · v ∈ vertL2 ⇒ tL2_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL2 ⇒ tL2_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL2 ⇒ tL2_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL2 ⇒ mV(sourceL2(e)) = sourceG(mE(e)) ∧ mV(targetL2(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL2 ⇒ mV(attrNL2(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
  @grd_NAC_E1 ¬(∃forbRT·
                  forbRT ⊆ edgeG \ mE[edgeL2] ∧ tG_E(forbRT) = RT ∧
                  sourceG(forbRT) = mV(T2_5) ∧ targetG(forbRT) = mV(T2_3))
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e2_8, e2_7}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e2_8 ↦ mV(T2_4), e2_7 ↦ mV(T2_5)}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e2_8 ↦ mV(T2_5), e2_7 ↦ mV(T2_3)}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e2_8 ↦ LockRT, e2_7 ↦ RT}
end
```

The events labelled as 1 through 5 cover the five steps of a *begin* operation, they can be found in Appendix C.1.

5.3.3 Read Operation

The following context and event represents a **read operation**, as seen in Figure 32. The context part for this rule describes the L6 graph used for the pattern match, and one interesting definition of this rule is the possibility to adapt the *TOp*-node's inheritance. This can be seen in the axiom *@axm_tL6_V_def* that contains the partition

definition of L6's vertices. In this partition, the vertex labelled as *T6_2* has multiple possibilities for types, so when the pattern match occurs, any of the mappings can satisfy as a correct type.

```
sets
  vertL6
  edgeL6
  attrL6
constants
  T6_1 T6_2 T6_3 T6_4 T6_5
  e6_1 e6_2 e6_3
  atr6_1 atr6_2 atr6_3 atr6_4
  sourceL6
  targetL6
  attrNL6
  attrVL6
  tL6_V
  tL6_E
  tL6_A
axioms
  @axm_vertL6 partition(vertL6, {T6_1}, {T6_2}, {T6_3}, {T6_4}, {T6_5})
  @axm_edgeL6 partition(edgeL6, {e6_1}, {e6_2}, {e6_3})
  @axm_attrL6 partition(attrL6, {atr6_1}, {atr6_2}, {atr6_3}, {atr6_4})
  @axm_sourceL6_type sourceL6 ∈ edgeL6 → vertL6
  @axm_sourceL6_def partition(sourceL6, {e6_1 ↦ T6_1}, {e6_2 ↦ T6_3}, {e6_3 ↦ T6_4})
  @axm_targetL6_type targetL6 ∈ edgeL6 → vertL6
  @axm_targetL6_def partition(targetL6, {e6_1 ↦ T6_3}, {e6_2 ↦ T6_2}, {e6_3 ↦ T6_4})
  @axm_attrNL6_type attrNL6 ∈ attrL6 → vertL6
  @axm_attrNL6_def partition(attrNL6, {atr6_1 ↦ T6_4}, {atr6_2 ↦ T6_3}, {atr6_3 ↦ T6_5}, {atr6_4 ↦ T6_4})
  @axm_attrVL6_type attrVL6 ∈ attrL6 → ℙ(ℕ)
  @axm_attrVL6_def partition(attrVL6, {atr6_1 ↦ ℕ}, {atr6_2 ↦ ℕ}, {atr6_3 ↦ ℕ}, {atr6_4 ↦ ℕ})
  @axm_tL6_V tL6_V ∈ vertL6 → vertT
  @axm_tL6_V_def partition(tL6_V, {T6_1 ↦ History}, {T6_2 ↦ Begin}, {T6_2 ↦ Read}, {T6_2 ↦ Write}, {T6_2 ↦ Commit},
                  {T6_2 ↦ Abort}, {T6_2 ↦ End}, {T6_3 ↦ Read}, {T6_4 ↦ CGNode}, {T6_5 ↦ CGNode})
  @axm_tL6_E tL6_E ∈ edgeL6 → edgeT
  @axm_tL6_E_def partition(tL6_E, {e6_1 ↦ OP}, {e6_2 ↦ Next}, {e6_3 ↦ Loc})
  @axm_tL6_A tL6_A ∈ attrL6 → attrT
  @axm_tL6_A_def partition(tL6_A, {atr6_1 ↦ From}, {atr6_2 ↦ ID}, {atr6_3 ↦ ID}, {atr6_4 ↦ ID})
```

The event for a *read* operation can be seen below. Not many transformations are needed for this operation, as it only needs to deal with edges. The *OP*-edge representing the current operation to be executed is "moved" to the next operation, pointing from *T6_3* to *T6_2*. This happens by deleting *e6_1* and creating *e6_4*. Lastly, a new *RF*-edge (*e6_7*) is created and added to the conflict graph nodes of the corresponding transactions.

```
event Read
any
  mV
  mE
  mA
  e6_4
  e6_7
  delE
where
```

```
  @grd_mV mV ∈ vertL6 → vertG
  @grd_mE mE ∈ edgeL6 → edgeG
  @grd_mA mA ∈ attrL6 → attrG
  @grd_delE delE = [{e6_1}]
  @grd_newe6_4 e6_4 ∈ ℕ\edgeG
  @grd_newe6_7 e6_7 ∈ ℕ\edgeG
  @grd_e6_4_e6_7 e6_4 ≠ e6_7
  @grd_vertices ∀v · v ∈ vertL6 ⇒ tL6_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL6 ⇒ tL6_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL6 ⇒ tL6_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL6 ⇒ mV(sourceL6(e)) = sourceG(mE(e)) ∧ mV(targetL6(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL6 ⇒ mV(attrNL6(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e6_4, e6_7}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e6_4 ↦ mV(T6_1), e6_7 ↦ mV(T6_5)}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e6_4 ↦ mV(T6_2), e6_7 ↦ mV(T6_4)}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e6_4 ↦ OP, e6_7 ↦ RF}
end
```

5.3.4 Write Operation

As for the **write operation** seen in Figure 33, the context and event follow a similar definition as the *read* operation. A pattern match using the *TOp*-node, and only deals with some edges and attributes when transforming the state. The event is set as follows. The same *OP*-edge is "moved" from *T7_3* to *T7_2*, by deleting *e7_1* and creating *e7_4*. As for the conflict graph's node, a new attribute *atr7_6* is created to represent which variable this transaction wrote to. This is used later to analyse writing conflicts during commit time. The value assigned to the new attribute uses the auxiliary function *attrVG* and the current matching *mA* to find the value of the already existing attribute *attr7_1*. This can be seen in action *@act_attrVG*.

```
event Write
any
  mV
  mE
  mA
  e7_4
  atr7_6
  delE
where
  @grd_mV mV ∈ vertL7 → vertG
  @grd_mE mE ∈ edgeL7 → edgeG
  @grd_mA mA ∈ attrL7 → attrG
  @grd_delE delE = [{e7_1}]
  @grd_newe7_4 e7_4 ∈ ℕ\edgeG
  @grd_newatr7_6 atr7_6 ∈ ℕ\attrG
  @grd_vertices ∀v · v ∈ vertL7 ⇒ tL7_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL7 ⇒ tL7_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL7 ⇒ tL7_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL7 ⇒ mV(sourceL7(e)) = sourceG(mE(e)) ∧ mV(targetL7(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL7 ⇒ mV(attrNL7(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e7_4}
  @act_attrG attrG := attrG ∪ {atr7_6}
```

@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e7_4 ↦ mV(T7_1)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e7_4 ↦ mV(T7_2)}
@act_attrNG attrNG := attrNG ∪ {atr7_6 ↦ mV(T7_4)}
@act_attrVG attrVG := attrVG ∪ {atr7_6 ↦ attrVG(mA(atr7_1))}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e7_4 ↦ OP}
@act_tG_A tG_A := tG_A ∪ {atr7_6 ↦ Writes}
end

5.3.5 Commit Operation

The first production rule, out of five, for a **commit operation** is called *CommitLock* and its can be seen below. This event represents the graph transformation seen in Figure 61. In this case, the only change made is to the *OP*-edge of the history, deleting it and creating a *Lock*-edge in its place.

event CommitLock
any
mV
mE
mA
e8_4
delE
where
@grd_mV mV ∈ vertL8 → vertG
@grd_mE mE ∈ edgeL8 → edgeG
@grd_mA mA ∈ attrL8 → attrG
@grd_delE delE = [{e8_1}]
@grd_newe8_4 e8_4 ∈ ℕ\edgeG
@grd_vertices ∀v · v ∈ vertL8 ⇒ tL8_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL8 ⇒ tL8_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL8 ⇒ tL8_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL8 ⇒ mV(sourceL8(e)) = sourceG(mE(e)) ∧ mV(targetL8(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL8 ⇒ mV(attrNL8(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e8_4}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e8_4 ↦ mV(T8_1)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e8_4 ↦ mV(T8_2)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e8_4 ↦ Lock}
end

The second step of a *commit* is the production rule called *CommitLoop1*, and the event for this rule can be seen below. As shown in the graph transformation in Figure 62, this rule serves as a loop through visible transactions that wrote to the same variable as the main transaction. The transformation of the state includes deleting the *Free*-edge (*e9_3*), creating a *LockC*-edge (*e9_8*) in its place. Also, creating a *WW*-edge between the visible transaction's conflict graph node and the main one. As a NAC, the write-before-write relation cannot already exist between the two conflict graph nodes, this is denoted in the guard *@grd_NAC_E1*.

event CommitLock_Loop1
any
mV
mE

mA
e9_7
e9_8
delE
where
@grd_mV mV ∈ vertL9 → vertG
@grd_mE mE ∈ edgeL9 → edgeG
@grd_mA mA ∈ attrL9 → attrG
@grd_delE delE = [{e9_3}]
@grd_newe9_7 e9_7 ∈ ℕ\edgeG
@grd_newe9_8 e9_8 ∈ ℕ\edgeG
@grd_e9_7_e9_8 e9_7 ≠ e9_8
@grd_vertices ∀v · v ∈ vertL9 ⇒ tL9_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL9 ⇒ tL9_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL9 ⇒ tL9_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL9 ⇒ mV(sourceL9(e)) = sourceG(mE(e)) ∧ mV(targetL9(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL9 ⇒ mV(attrNL9(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
@grd_NAC_E1 ¬(∃forbWW·
forbWW ⊆ edgeG \ mE[edgeL9] ∧ tG_E(forbWW) = WW ∧
sourceG(forbWW) = mV(T9_5) ∧ targetG(forbWW) = mV(T9_3))
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e9_7, e9_8}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e9_7 ↦ mV(T9_5), e9_8 ↦ mV(T9_4)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e9_7 ↦ mV(T9_3), e9_8 ↦ mV(T9_5)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e9_7 ↦ WW, e9_8 ↦ LockC}
end

The events labelled as 8 through 12 cover the five steps of a *commit* operation, they can be found in Appendix C.2.

5.3.6 Abort Operation

The last transactional operation is an **abort operation**. As seen in Figure 35, an *abort* adds a *Done*-edge (*e13_6*) to the conflict graph node for this transaction, denoting that it finished but it's not visible to other transactions. Moreover, it also "moves" the *OP*-edge to the next operation by deleting *e13_1* and creating *e13_4*.

event Abort
any
mV
mE
mA
e13_4
e13_6
delE
where
@grd_mV mV ∈ vertL13 → vertG
@grd_mE mE ∈ edgeL13 → edgeG
@grd_mA mA ∈ attrL13 → attrG
@grd_delE delE = [{e13_1}]
@grd_newe13_4 e13_4 ∈ ℕ\edgeG
@grd_newe13_6 e13_6 ∈ ℕ\edgeG
@grd_e13_4_e13_6 e13_4 ≠ e13_6
@grd_vertices ∀v · v ∈ vertL13 ⇒ tL13_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL13 ⇒ tL13_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL13 ⇒ tL13_A(a) = tG_A(mA(a))

@grd_srctgt $\forall e \cdot e \in$ edgeL13 $\Rightarrow$ mV(sourceL13(e)) = sourceG(mE(e)) $\wedge$ mV(targetL13(e)) = targetG(mE(e))
@grd_attrComp $\forall a \cdot a \in$ attrL13 $\Rightarrow$ mV(attrNL13(a))=attrNG(mA(a)) $\wedge$ attrVG(mA(a)) $\in \mathbb{N}$
then
@act_edgeG edgeG := (edgeG $\setminus$ delE) $\cup$ {e13_4, e13_6}
@act_sourceG sourceG := (delE $\lhd$ sourceG) $\cup$ {e13_4 $\mapsto$ mV(T13_1), e13_6 $\mapsto$ mV(T13_4)}
@act_targetG targetG := (delE $\lhd$ targetG) $\cup$ {e13_4 $\mapsto$ mV(T13_2), e13_6 $\mapsto$ mV(T13_4)}
@act_tG_E tG_E := (delE $\lhd$ tG_E) $\cup$ {e13_4 $\mapsto$ OP, e13_6 $\mapsto$ Done}
end

5.3.7 Conflict Graph

After executing all the transactional operations, all that is left is to evaluate the conflict graph. The main goal of the evaluation is to look for any loop in the conflict graph, and this is achieved via a simple path marking of visited nodes. The methodology is to start from the transaction with an attribute *ID* of 0 and take steps following every directed edge, leaving tokens behind marking the nodes as "already visited". A step cannot be taken only if there are no more directed edges to follow, or there is already a token in the same place. This serves as a stopping point no matter how large the conflict node may be. If the situation happens where there is a "step" and a "token" in the same node, this denotes a cycle in the conflict graph.

To implement this evaluation of cycles, two production rules are needed. The first, to start this pathing is named *LoopStart*, as seen in Figure 36. This transformation deletes the *OP*-edge from the history, therefore no more transactional operations can execute, and adds a *LoopStep*-edge to the conflict node with an attribute of *ID* = 0.

event LoopStart
any
mV
mE
mA
e14_2
delE
where
@grd_mV mV $\in$ vertL14 $\rightarrow$ vertG
@grd_mE mE $\in$ edgeL14 $\rightarrow$ edgeG
@grd_mA mA $\in$ attrL14 $\rightarrow$ attrG
@grd_delE delE = [{e14_1}]
@grd_newe14_2 e14_2 $\in \mathbb{N} \setminus$ edgeG
@grd_vertices $\forall v \cdot v \in$ vertL14 $\Rightarrow$ tL14_V(v) = tG_V(mV(v))
@grd_edges $\forall e \cdot e \in$ edgeL14 $\Rightarrow$ tL14_E(e) = tG_E(mE(e))
@grd_attrs $\forall a \cdot a \in$ attrL14 $\Rightarrow$ tL14_A(a) = tG_A(mA(a))
@grd_srctgt $\forall e \cdot e \in$ edgeL14 $\Rightarrow$ mV(sourceL14(e)) = sourceG(mE(e)) $\wedge$ mV(targetL14(e)) = targetG(mE(e))
@grd_attrComp $\forall a \cdot a \in$ attrL14 $\Rightarrow$ mV(attrNL14(a))=attrNG(mA(a)) $\wedge$ attrVG(mA(a)) $\in \mathbb{N}$
then
@act_edgeG edgeG := (edgeG $\setminus$ delE) $\cup$ {e14_2}
@act_sourceG sourceG := (delE $\lhd$ sourceG) $\cup$ {e14_2 $\mapsto$ mV(T14_3)}
@act_targetG targetG := (delE $\lhd$ targetG) $\cup$ {e14_2 $\mapsto$ mV(T14_3)}
@act_tG_E tG_E := (delE $\lhd$ tG_E) $\cup$ {e14_2 $\mapsto$ LoopStep}
end

The second production rule needed is the one that does all the marking, and its called *LoopStep*, as seen in Figure 37. The graph transformation includes deleting the

current *LoopStep*-edge (*e15_1*) and creating a new one *e15_5*. At the same time it also creates the *LoopToken*-edge (*e15_3*) in the first *CGNode*. The NAC used for the pattern match is that there should not be already a *LoopToken*-edge (*e15_6*) in the first conflict graph node.

event LoopStep
any
  mV
  mE
  mA
  e15_3
  e15_5
  delE
where
  @grd_mV mV ∈ vertL15 → vertG
  @grd_mE mE ∈ edgeL15 → edgeG
  @grd_mA mA ∈ attrL15 → attrG
  @grd_delE delE = [{e15_1}]
  @grd_newe15_3 e15_3 ∈ ℕ\edgeG
  @grd_newe15_5 e15_5 ∈ ℕ\edgeG
  @grd_e15_3_e15_5 e15_3 ≠ e15_5
  @grd_vertices ∀v · v ∈ vertL15 ⇒ tL15_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL15 ⇒ tL15_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL15 ⇒ tL15_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL15 ⇒ mV(sourceL15(e)) = sourceG(mE(e)) ∧ mV(targetL15(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL15 ⇒ mV(attrNL15(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
  @grd_NAC_E1 ¬(∃forbLoopToken·
                forbLoopToken ⊆ edgeG \ mE[edgeL15] ∧ tG_E(forbLoopToken) = LoopToken ∧
                sourceG(forbLoopToken) = mV(T15_1) ∧ targetG(forbLoopToken) = mV(T15_1))
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e15_3, e15_5}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e15_3 ↦ mV(T15_1), e15_5 ↦ mV(T15_2)}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e15_3 ↦ mV(T15_1), e15_5 ↦ mV(T15_2)}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e15_3 ↦ LoopToken, e15_5 ↦ LoopStep}
end

With this, the invariant seen in the initial state can be applied to the system and if at all reachable states of the system no cycles are found, the history indeed does satisfy the correctness criterion of opacity.

5.4 Discussion

The event-B model of a Graph Grammar (GG) includes mainly a type graph, an initial state and a set of transformation rules. This defines how the state of the system can behave based on the initial state and whether or not the events can find their respective pattern match. This pattern match is already a form of restriction for when to transform states, meaning that the set of conditions in an event could potentially end the execution prematurely. Therefore, in this application of the event-B translation of a GG model, where a software transactional memory (STM) system is implemented, the main properties of the system are not present in an event's guards but instead in the initial state's invariants. Thus, these properties can be tested and potentially proved for

every reachable state of the system, which should always include the entire history of transactional memory operations.

When translating the states of a graph grammar into the event-B model, the result is a finite number of sets that contain unique elements, and functions on how the connections between elements are mapped. To analyse any properties of the system, is to analyse how these sets are composed and which elements and relations are present or not. For this work, where a correctness criterion test of an STM history is the goal, it becomes necessary the construction and analysis of a conflict graph that represents how each transaction interact with each others operations. These interactions can be a read of a variable, an overwrite, a combination of the two resulting in an inconsistency, or even just the fact that the first operation of a transaction happens after all other operations of another transaction. In the case of a conflict graph, the acyclicity property can be easily derived from the use of the production rule *LoopStep* that marks the conflict graph nodes with self-edges that represent a path taken by following the directed edges.

As seen before, the goal of the *LoopStep* production rule is to follow the directed edges and mark the conflict graph nodes with *LoopStep*-edges, at the same time leaving *LoopToken*-edges behind. So if the situation where a *LoopStep* and *LoopToken*-edge ever happen, it means that there is cycle in the conflict graph. In the event-B model, the test for this particular situation is to use an invariant declared in the initial state, this way any reachable state will be tested for this condition. The code below shows the invariant for this property of acyclicity. Essentially what it does is to look at every two edges *e1, e2* in the set of edges (*edgeG*), if their type are of step and token (*tG_E*) as the property requires, the mapping for their source vertex (in *sourceG*) cannot be the equal.

$$\text{@inv\_propAcyclic: } \forall e1,e2 \cdot e1 \in edgeG \land tG\_E(e1) = LoopToken \land e2 \in edgeG \land tG\_E(e2) = LoopStep \implies sourceG(e1) \neq sourceG(e2)$$

Besides the acyclicity property that reasons about an external condition of logical correctness of the system, invariants can also be used to enhance the guarantee of structural correctness of the system as it is transformed by events. The same idea of looking into the sets of elements, mappings and types defined in the initial state can be applied to set extra conditions that the type graph may not be able enforce. For example, the type graph only denotes that a *Next*-edge can be used between transactional operation vertices (begin, read, write, etc). What the type graph does not enforce is, if a *Next*-edge has to map to two distinct vertices of the “correct type”, technically there could be a *Next*-edge whose source and target map to the same vertex. The invariant shown below could be used to prevent this, and because it is set up at the initial state, every reachable state guarantees this correctness.

$$\text{@inv\_propNext: } \forall e \cdot e \in edgeG \land tG\_E(e) = Next \implies sourceG(e) \neq targetG(e)$$

Some of the edges used in this work always have their source and target as equal values (*Vis*, *Done*, *LoopStep*, *LoopToken*, etc), because they are used as flags for conditions of a specific element and never to connect two vertices. It is also for the benefit of the system that they keep this property, and this can be implemented in the same way as acyclicity by using invariants. Other properties of the structure of the system are: the edges representing transactional relations (*RW*, *RF*, *RT* and *WW*) never have equal source and target; a "single file" history, where there are only one *Next*-edge in and out of each operation node; transactions have unique identifiers as attributes; every write operation uses unique values, and more.

By the nature of an event-B model and its invariants, the system can be made to guarantee properties at each step, this is interesting for a full implementation of an algorithm. In this work, only a single history was evaluated as an initial state, but as demonstrated in Chapter 6 the graph transformation approach on software transactional memory algorithms can generate histories dynamically. An event-B model of the full algorithm could be used as a more concrete proof of correctness, because it deals with all reachable states and is less limited by the initial state as these previous approaches.

Lastly, although the complexity of a transactional operation can be a challenge that the graph transformation portion of the approach has to overcome, if it is possible to create the proper production rules, the event-B translation is rather direct. Besides using natural numbers as identifiers for nodes, edges, and attributes, all functionalities of a production rule can be described using axioms and events. Even negative application conditions are easily translated as guards inside events, because most elements of the NAC graph is actually part of the pattern matched L side of the production rule, only a few extra guards are enough to add these negative conditions. Any extra condition on the logical and structural side of the system can be added as invariants, to be tested for every state, or as a guard, to be tested for a specific event/production rule.

6 APPLICATIONS FOR THE GG APPROACH

This chapter describes the two transactional memory algorithms used as case studies for the graph transformation formalism described in this thesis. The first algorithm is the CaPR+ (ANAND; SHYAMASUNDAR; PERI, 2016), it presents a balance between novelty and complexity as it deals with checkpoints and partial rollbacks. The original definition also includes the same graph characterization of opacity tackled in this thesis, which is used as proof of correctness. The second algorithm described in this chapter is the STM library for Haskell (HARRIS et al., 2005), a more well known algorithm that does not satisfy the property of opacity. Section 6.1 presents the CaPR+ algorithm and how the graph transformation approach deals with its complexity and correctness proof. Section 6.2 presents the STM Haskell algorithm, its retry functionality and some analysis on the correctness test of the algorithm.

6.1 CaPR+ algorithm

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 Grammar.[1] The choice is supported by the fact that this algorithm has a more complex logic than other traditional TM algorithms such as TL2 (DICE; SHALEV; SHAVIT, 2006) and SwissTM (DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009), 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 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 algorithm provides a natural way to realize a hybrid system of pure aborts and partial rollbacks.

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

[1] Full code available in `https://github.com/diogocrds/Thesis`

workspace are as follows:

- Local Data Block (LDB, Figure 38(a)): Each entry consists of the local object and its current value in the transaction;
- Shared Object Store (SOS, Figure 38(b)): 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, Figure 38(c)): 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.

Object	Value

(a) Local Data Block (LDB)

Object	Current Value	Read Flag	Write Flag

(b) Shared object Store (SOS)

Victim Shared Object	Program Location	Local Snapshot

(c) Checkpoint Log (Cplog)

Figure 38 – Local workspace for CaPR+ algorithm.

The data structures in the global workspace are:

- Global List of Active Transactions (Actrans, Figure 39): 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.

Transaction ID	Status Flag	Conflict Objects

Figure 39 – Global List of Active Transactions (Actrans)

The full CaPR+ algorithm can be seen Appendix A.1.

Figure 40 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 graph grammar.

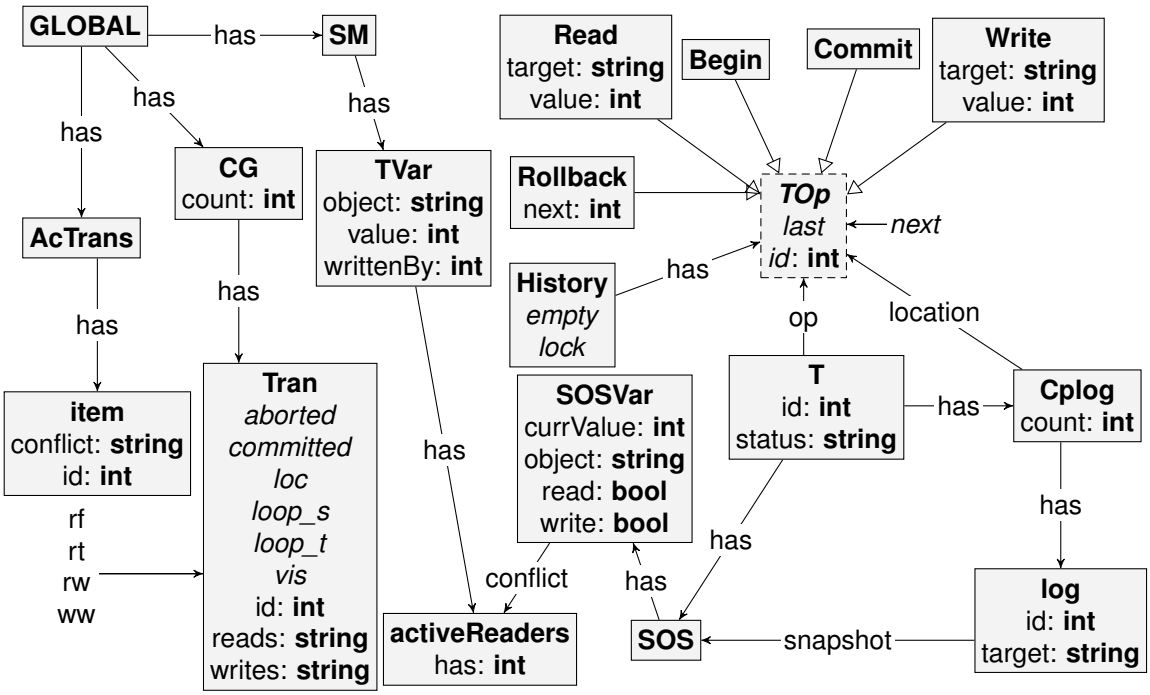


Figure 40 – Type graph of CaPR+ algorithm.

In the CaPR+ algorithm a read operation can be used in two situations: reading a t-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 41 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 (*SOSVar* that stores the current value from *TVar*); a checkpoint for a possible partial rollback in the future (*Cplog* now has an edge *location* to the current *Read* operation); a snapshot of the current state of the transaction to rollback to (*Cplog* now has a *log* that stores a copy of the SOS and which variable triggered the check-

point). 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 “reads-from” relation between the transaction that wrote the value being read and the one executing this read operation.

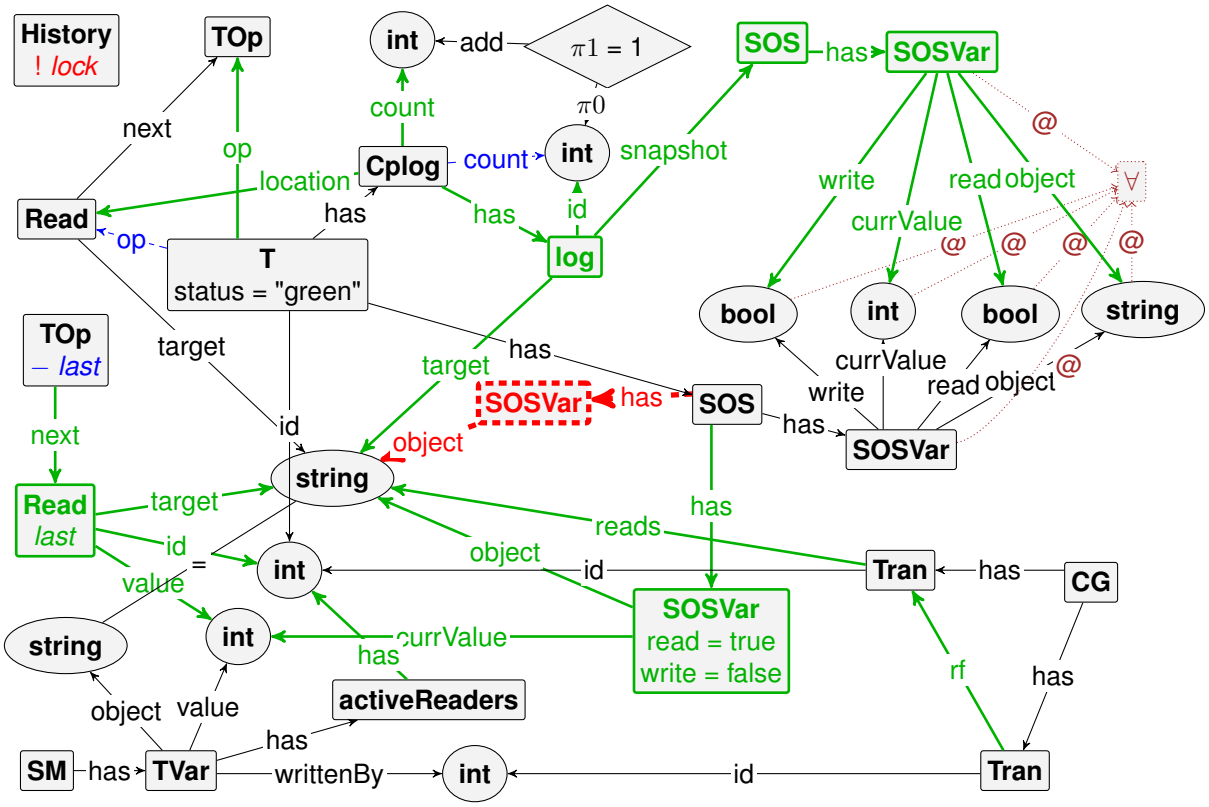


Figure 41 – Read from Shared Memory operation of CaPR+ algorithm.

Another production rule is used to read a variable on a local access via the SOS object, this is shown in Figure 42. Because the *SOSVar* already exists, a local read only needs to use its stored current value. The *op* edge moves to the next operation and a new object is created at the last position of the history (*Read* with the flag *last*). Any write operation is always done locally, so write operations only manipulate *SOSVar*-nodes and never deal directly with the shared memory. There is also the case where the local copy does not exist yet but a write operation is called, this is dealt the same way as a read operation: two separate rules, one for when the local copy does not exist and one for when it does. In the original definition by Anand; Shyamasundar; Peri (2016), the algorithm does not have an *abort* operation, every transaction that tries to commit is either successful or has to rollback to the safest checkpoint (earliest conflicting read operation to the SM). 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, which are mutually exclusive. A commit can originate 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.

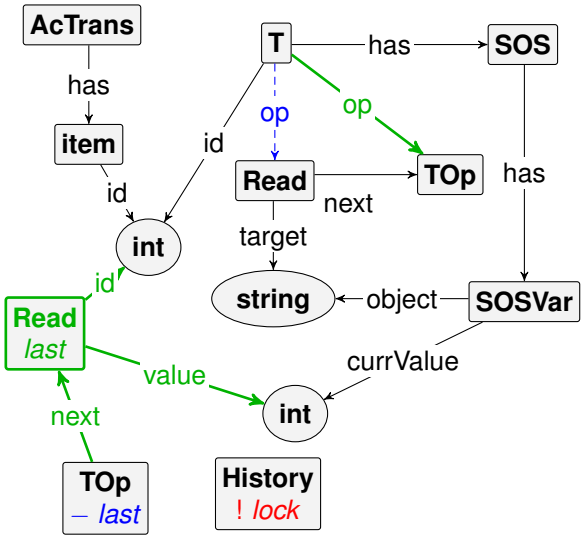


Figure 42 – Read from local copy operation of CaPR+ algorithm.

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.

Figure 43 shows one of the cases for the rollback operation, when there is only one checkpoint so the rollback is direct. In the case of multiple checkpoints more conditions are used to ensure the correct state to rollback to (which variable generated the conflict versus which one is safe). In the case of a single checkpoint, the *Cplog* shows a counter of 1 and an *location* edge to the correct read operation used as the next inline. The current transaction (*T* node) is flagged as “green” again, its old identifier is deleted and replaced with a new one, the old SOS is deleted and replaced with the one stored on the *log*’s snapshot. The transaction being rolled back cannot be an active reader or have any active conflicts anymore so any of those notations are also deleted. Finally, a new *Tran* node is added to the CG with the new identifier, while the old one is flagged as “aborted”.

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. The result

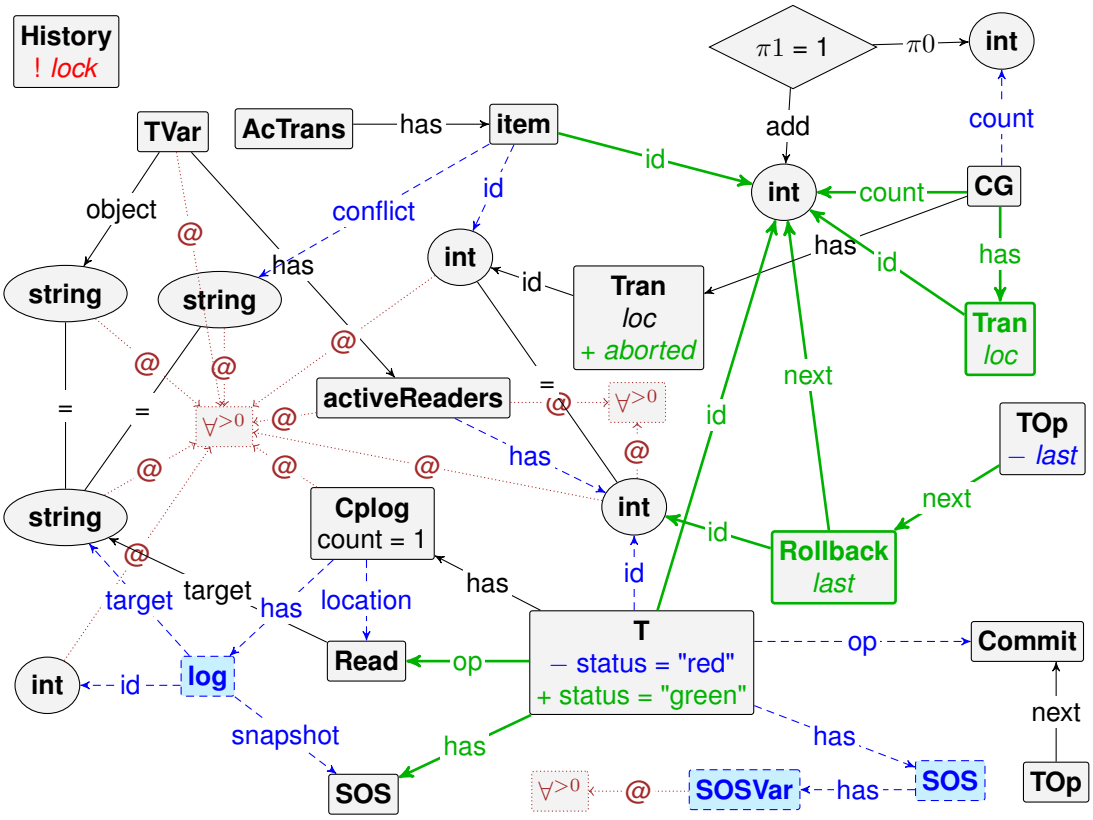


Figure 43 – Rollback operation for one checkpoint of CaPR+ algorithm.

is that any subsequent 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.

Figure 44 shows an example of transaction code with two conflicting transactions and history H_2 resulted from an execution of this code. Transaction T_2 performs a rollback when the second read operation was supposed to be called, at that point the system recognizes the inconsistent values that were read from a previous operation. A new transaction T_3 is created and executes the operations from T_2 with a consistent state of the shared memory. The conflict graph for this execution can be seen in Figure 45. It can be noticed that if the inconsistent state had not been dealt with at second read operation on T_2, and instead at commit time, it would be reflect on the CG via an edge of reads-from "rf" from T_1 to T_2 (node *Tran* with id 1 and 2, respectively) resulting in a cycle and reflecting that opacity is satisfied in this execution.

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 t-variable, but also has the real-time relation between transactions. It was observed that with three transactions that read and write to

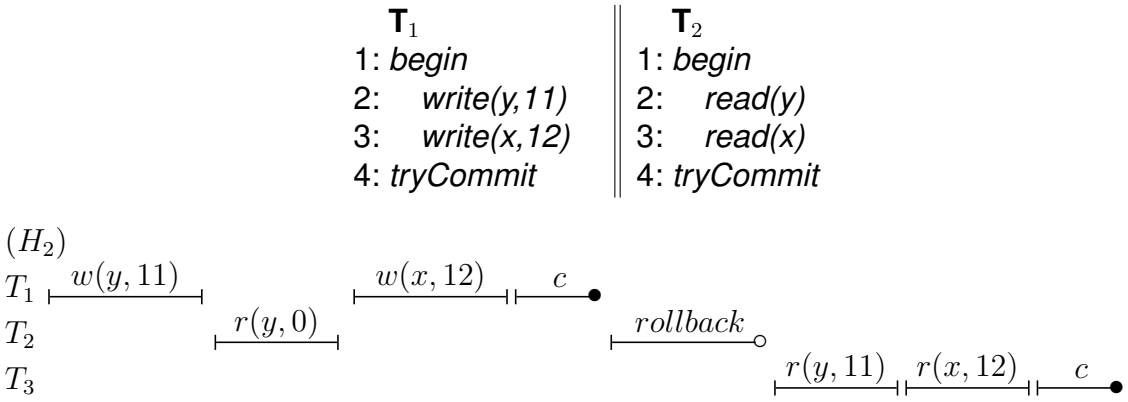


Figure 44 – Example of transaction code and opaque history.

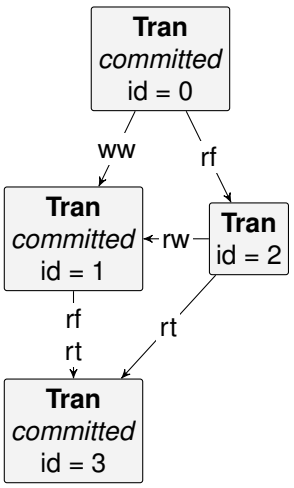


Figure 45 – Conflict Graph for opaque history.

the same t-variable (similar to Figure 13), 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. A conclusion that can be extracted from this is the number of conflicts has no influence on ability to deal with them, because as said before the decision making of how the rules are executed will process every conflict individually but 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 a combination of conflicts (from read, write and real-time relations) that is complete enough that any bigger input would just result in some redundant computations.

Using this approach, the translation of the CaPR+ algorithm to a graph grammar managed to generate only acyclic conflict graphs, proving that the algorithm only generates opaque histories. This is formalized by the GG of the algorithm, which includes a group of production rules and an initial state, and the CTL formula used to check for acyclicity of the conflict graph. The results of the CaPR+ state space indicated that optimization of the LTS can be relevant, in its current state the graph grammar can

generate over 700.000 unique histories with an input of three transactions. Considering each history has to generate a few more states for the acyclicity test, it can result in a large number of states. However, with some modifications to the production rules it is possible to take advantage of the fact that different histories may generate the same conflict graph, so the paths of the LTS converge, lowering the number of states generated.

6.2 STM Haskell algorithm

This section describes the graph grammar developed to evaluate correctness of the STM library for Haskell (HARRIS et al., 2005). One of the characteristics of STM Haskell is that it allows for transactions to read inconsistent values from the shared memory, this can lead to an undesirable situation of indefinite loops. To avoid this, every time the scheduler is called to switch to a thread that is engaged in a transaction, the scheduler first calls a validation method to check that the transaction is not already doomed (HARRIS et al., 2005; MARLOW; PEYTON JONES; SINGH, 2009).

The formalization using graph grammar[2] includes an initial state, a type graph, a set of production rules that transform the state to generate histories and the graph conditions for the correctness evaluation. The STM Haskell algorithm used can be found in Appendix A.2. Later in this section, a second algorithm is used for comparison in how decision making influences correctness, called TL2 (DICE; SHALEV; SHAVIT, 2006), and the algorithm code can be found in Appendix A.3.

The initial state consists of a set of two transactions that will purposefully generate inconsistent states for the correctness test to deal with, the empty history list, and the shared memory where transactional variables are already initiated. Figure 46 shows these components in more detail.

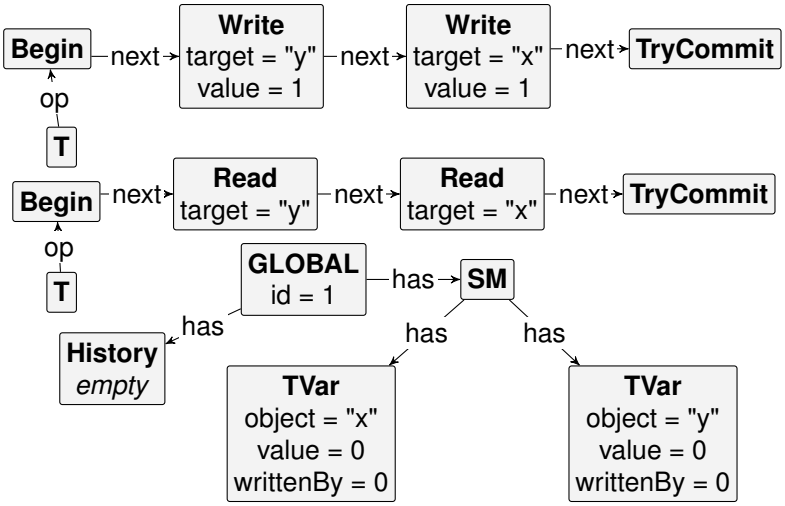


Figure 46 – Initial state of two transactions, empty history and shared memory for the GG.

[2]Full code available in `https://github.com/diogocrds/Thesis`

From this initial state the production rules now have to evaluate each transactional action and modify the shared memory accordingly. In the case of STM Haskell, all the modifications are first made in a local copy of the targeted variable. After that, should the commit operation be successful, the changes will only then be made in the shared memory objects that are visible to other transactions.

6.2.1 Type Graph

A type graph is used to not only guarantee the correct evolution of the system's states, but also to facilitate some of the definitions in the production rules. More specifically the inheritance relation between nodes that the GROOVE tool provides. In this GG, a Transactional Operations (TOp) is separated from History Operations (HOp) using this inheritance functionality.

Figure 47 shows the complete proposed type graph for the STM Haskell GG.

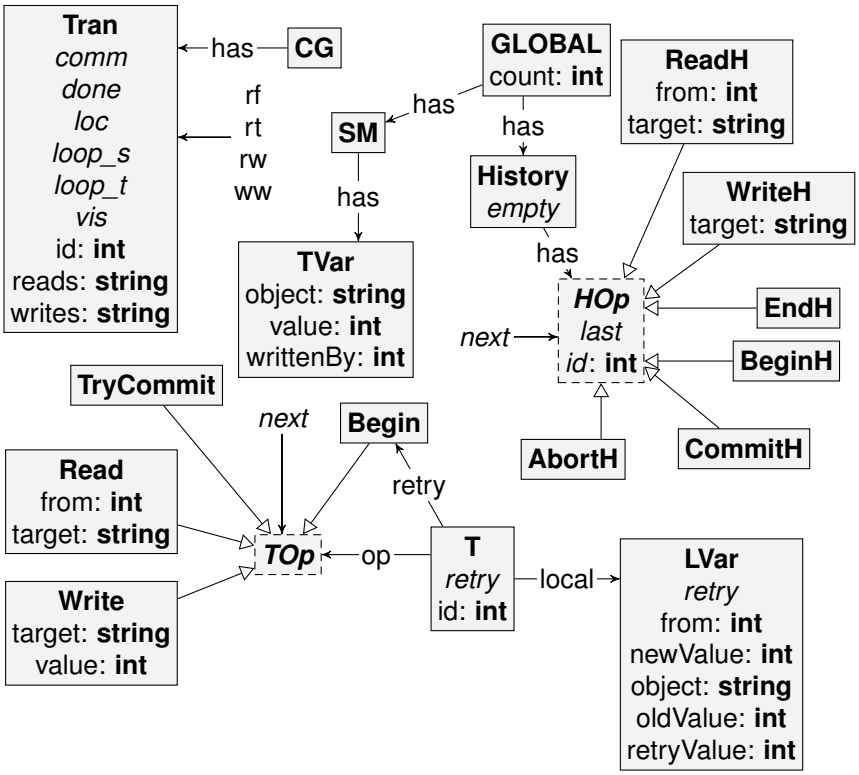


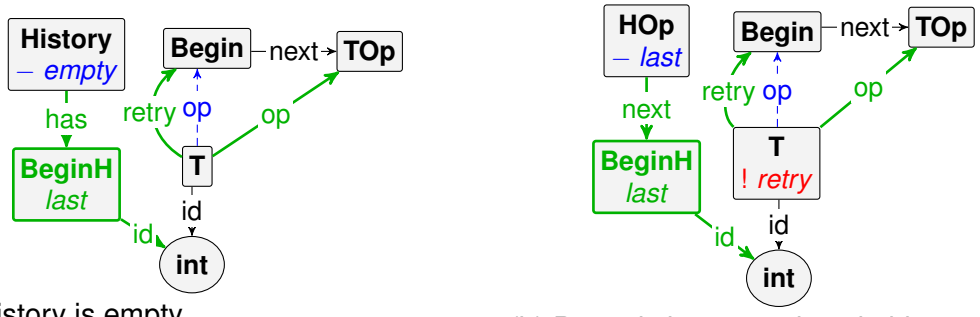
Figure 47 – Type Graph for the STM Haskell GG.

The type graph includes the transactional operations (*TOp*-nodes), that are always linked to an identified transaction node (*T*-node). The execution of these transactional operations modify the transactional log of its parent transaction, represented in the form a "local buffer" *LVar* that stores the old value of the targeted variable, and any new ones resulted from executing the operations. The history operations (*HOp*-node) are created as the transactional operations are executed, and they record information regarding the relations between transactions (which variable is being written or from which transaction a variable was read). Lastly, the type graph also ensures that the

Conflict Graph (CG) is also well constructed during the correctness verification.

6.2.2 Production Rules

The four main transactional operations that the proposed grammar presents are: *Begin*, *Read*, *Write* and *TryCommit*. In an ideal scenario, a grammar portraying the STM Haskell algorithm will have one production rule for each desired operation, however some of the logic in these operations can be complex and it is actually easier to split it into separate production rules. An example can be seen in Figure 48.



(a) History is empty.

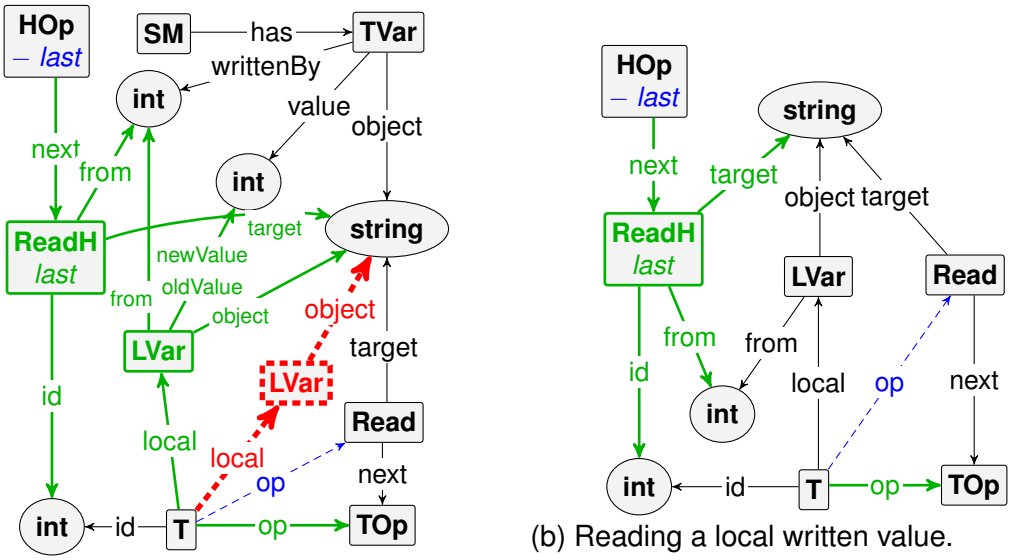
(b) Pre-existing operations in history.

Figure 48 – Production rules for a *Begin* operation.

A **Begin** operation can happen in two instances: at the start of the history (when it is empty), or in the middle of an already existing history. The production rule shown in Figure 48(a) demonstrates the *Begin* operation at the start of the history (top left node is a *History*-node flagged as “empty”), and in Figure 48(b) is the production rule for when the history is already initiated (top left node is a generic *HOp*-node flagged as “last”). In both production rules the result is the creation of a new *BeginH*-node flagged as the new “last”, and the attribution of an *id* to the transaction in question. The “retry” flag and edge shown here will be discussed in Section 6.2.3.

For every transaction, the first **Read** operation in STM Haskell creates a transactional log to be managed for the sake of correctness. The structure of the log is given by a collection of *LVar*-nodes connected to the transaction *T*-node via *local*-edges. The log stores the original value for each variable (*oldValue*-edge) and any subsequent values read or written (both happen locally after the first read, stored via the *newValue*-edge). Figure 49 shows the two production rules for a read operation. Figure 49(a) shows a production rule that executes when a transaction is reading from a variable for the first time, so it creates an *LVar*-node for the transaction storing the value read as both “old” and “new”. Figure 49(b) shows a production rule that executes when the *LVar*-node of the variable already exists, implying that the variable was already read or written beforehand.

The **Write** operation is set up in a similar way as a *Read* operation, Figure 50 shows both production rules for writing a value in a transactional variable. The first instance of a write operation uses the production rule shown in Figure 50(a), it requires that a

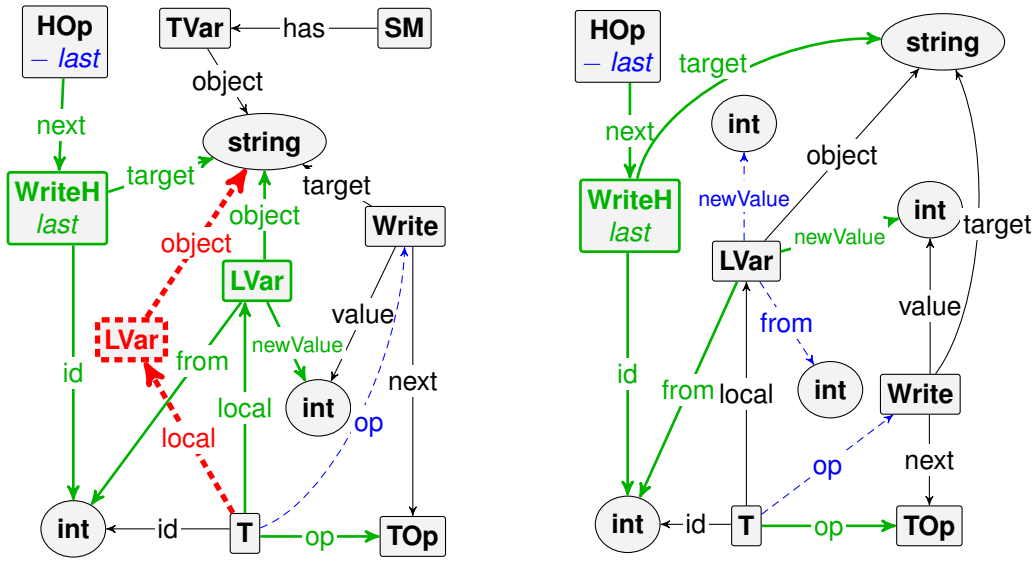


(a) Reading a value for the first time.

(b) Reading a local written value.

Figure 49 – Production rules for a *Read* operation.

local buffer (transactional log) for that variable does not exist, so it can create it. For the case that a local buffer already exists, Figure 50(b) is used. In both executions, the target value is stored locally and a new entry to the history is created.



(a) Writing a value for the first time.

(b) Writing a local value.

Figure 50 – Production rules for a *Write* operation.

So far, the allowed operations can initiate a transaction, read, and write variables from the shared memory. The last step is to make a decision as to whether the transaction can commit or not. The production rules responsible for this decision, in the case of STM Haskell, use the “old value” from each transactional variable that was read, to make a comparison to the current value in the shared memory. If the values of at least one variable is inconsistent, the transaction aborts. If no inconsistencies exist, the transaction commits. Figure 51 shows a production rule for a successful validation

and commit of a transaction. In the case of a failed verification, the production rule in Figure 52 is used instead, aborting the transaction. Some of the main differences in the case of an abort operation are: no manipulation of the shared memory is needed, because the log is local and is simply discarded; there must be at least one variable that was read and has an inconsistent state with the shared memory; and instead of just finalizing the execution, the transaction is flagged to "retry".

It is worth noting that Figure 51 shows the use of quantifiers to ensure that no inconsistencies appear in the shared memory (node *item* in *SM*). The notation can be read as: for all local *LVars* in the current transaction ($T \xrightarrow{local} LVar$) there exists a correspondent *TVar* in the shared memory ($SM \xrightarrow{has} TVar$) with the same object attribute. In that list of variables, there must not exist any local variable with and edge *oldValue* storing a different integer from the value stored in the shared memory.

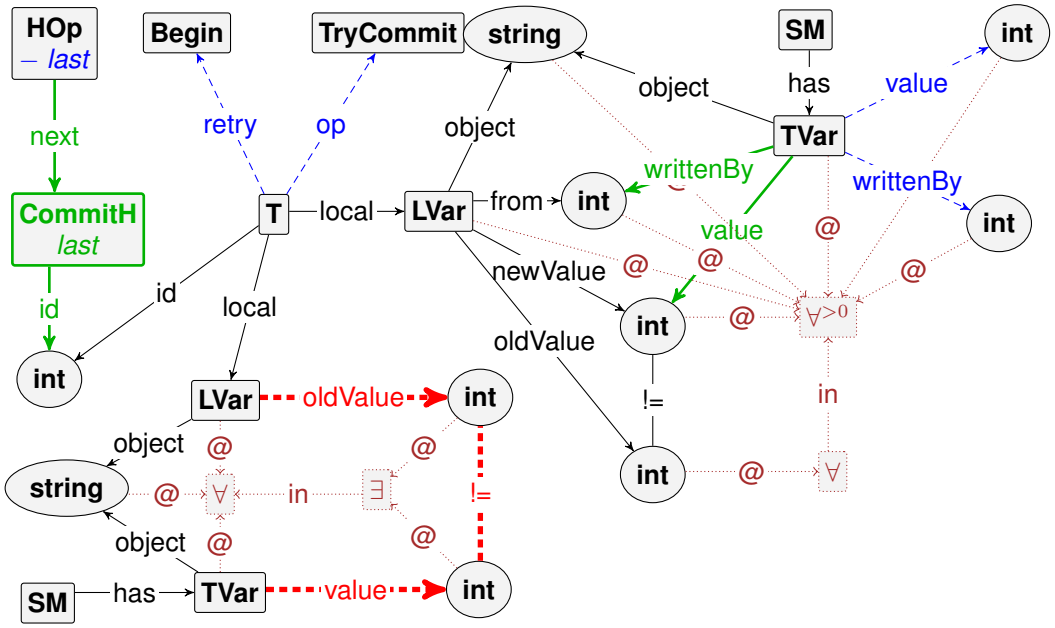


Figure 51 – *Commit* for the STM Haskell GG.

6.2.3 Retry Functionality

The retry functionality in the STM Haskell library does not behave in the expected way: to simply start the transaction from the beginning with a blank state when an abort occurs. When the verification fails and a transaction is flagged to restart, it does not immediately start executing again. Instead, the transaction waits for an update in a variable it performed a conflicting read operation.

Figure 53 shows the production rule that is executed when a transaction is triggered to restart by an update to a variable the transaction has read, generating an inconsistent state. This operation is very similar to a *Begin* operation shown in Figure 48(a). However, this time it requires the transaction to be flagged as "retry", and already existing local items (log) to compare incoming updates. The result of this production rule

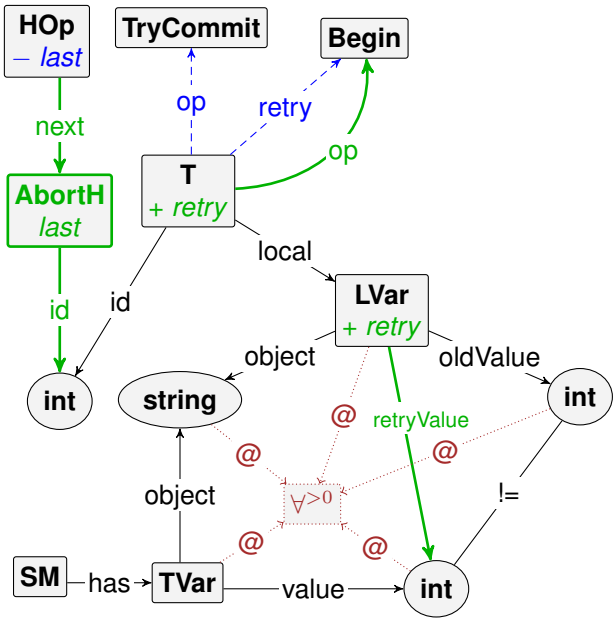


Figure 52 – *Abort* for the STM Haskell GG.

is the restart of the transaction with a new id and an empty log.

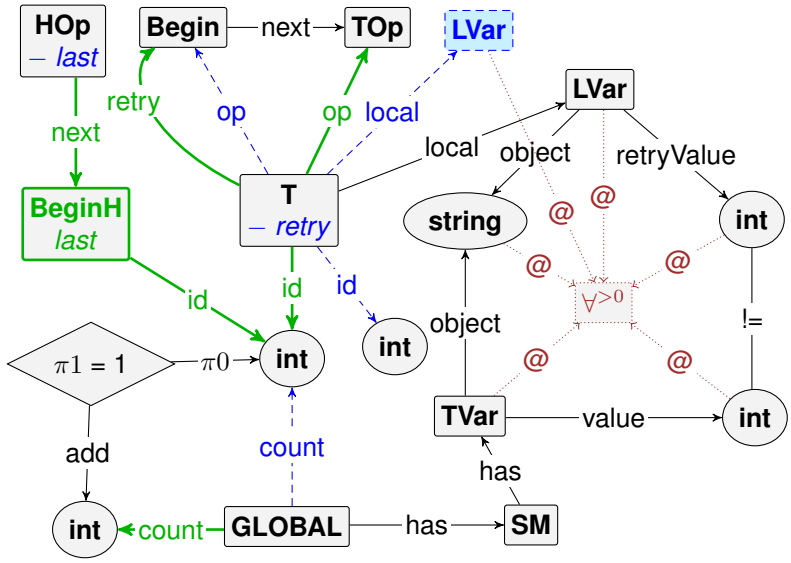


Figure 53 – *Begin* operation when a retry is triggered by an update.

6.2.4 Correctness Criterion

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 multi-

processor architecture, and so it should not limit the parallelism of transactions executed by different processes. A 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 (a TM where conflict is only checked at commit time).

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.

The work of 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.

As a subclass of Opacity, *Conflict Opacity* (CO-Opacity) ensures that every serialization respects the *conflict order* (WEIKUM; VOSSEN, 2001, Ch. 3). Let $Rset(T)$ and $Wset(T)$ be the sets of read and write operations of transaction *T*. For two transactions T_k and T_m in history *H*, it is said that T_k *precedes* T_m in conflict order, denoted $T_k \prec_H^{CO} T_m$, if:

- (w-w order) *Commit*$_k <_H$ *Commit*$_m$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$
- (w-r order) *Commit*$_k <_H$ *Read*$_m(x, v)$, $x \in Wset(T_k)$ and $v \neq$ Abort
- (r-w order) *Read*$_k(x, v) <_H$ *Commit*$_m$ and $x \in Wset(T_m)$ and $v \neq$ Abort

CO-Opacity differs from Opacity because it only deals with committed transactions (as opposed to including commit-pending transactions), so any conflict in aborted transactions is ignored in the conflict graph.

Definition 22. *A history H is said to be conflict opaque or co-opaque if H is valid and there exists a sequential legal history S such that (1) S is equivalent to Complete(H) and (2) S respects* $\prec_H^{RT}$ *and* $\prec_H^{CO}$*.*

Given a history *H*, a *conflict graph CG(H) = (V, E)* is constructed as follows: (1) *V* contains the set of transactions in *H*; (2) an edge (T_i, T_j) is added to *E* whenever $T_i \prec_H^{RT} T_j$ or $T_i \prec_H^{CO} T_j$.

Theorem 2. *A legal history H is co-opaque iff CG(H) is acyclic.*

Proof. Proof can be found in Kuznetsov; Peri (2017). □

6.2.5 Correctness Analysis

The correctness analysis performed on the proposed graph grammars involve generating a set of histories and comparing the conflict graphs based on two correctness criteria: Opacity (GUERRAOUI; KAPAŁKA, 2010) and CO-Opacity (KUZNETSOV; PERI, 2017). An auxiliary implementation a GG for the TL2 algorithm was used to compare the number of states generated, how conflict is detected and possible differences reflected in the conflict graph. Figure 54 shows an example of history that can be achieved from the initial state viewed in Figure 46 using the proposed STM Haskell GG. The history shows that STM Haskell aborts transaction T_2 and re-executes it as T_3 with correct read operations.

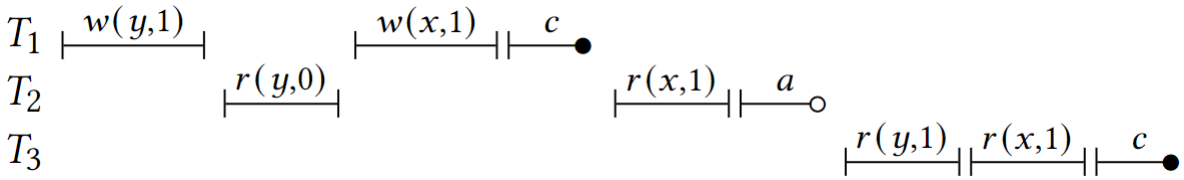
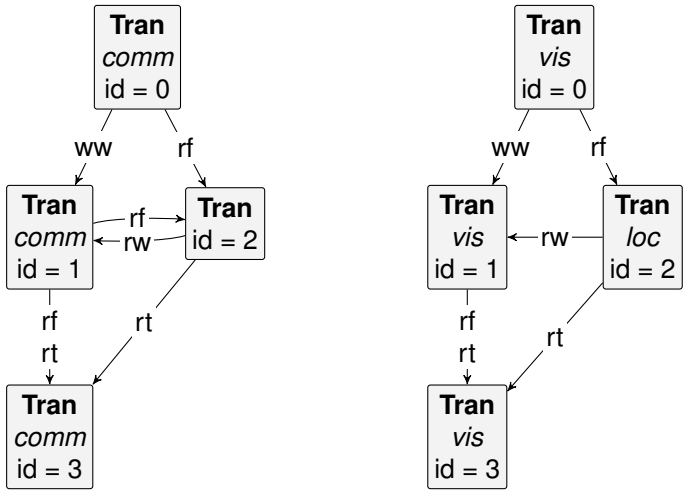


Figure 54 – History H_1 generated by STM Haskell.

From the definition of Opacity, history H_1 is not opaque because the aborted transaction T_2 performs read operations with inconsistent values. Neither $T_1T_2T_3$ nor $T_2T_1T_3$ are valid serializations of H_1. When evaluating for CO-Opacity, the consistency is taken from the point of view of each transaction locally. Meaning that the conflict graph is created using only prior committed transactions. In the case of H_1, the STM Haskell does not satisfy the correctness criteria. The problem derives from the same transaction T_2 reading inconsistent values, because T_1T_2 and T_2T_1 are both not valid serialization of local histories of T_2 in H_1.

Figure 55 shows the conflict graph generated by STM Haskell and TL2 following the execution that resulted in H_1. It is notable that in the case of TL2, the second read of T_2 $(r(x, 1))$ would not execute, and abort immediately instead. The prevention of this execution is reflected in the conflict graph, as the read operation would create a "reads from" edge (*rf*) between transactions T_1 and T_2. This additional edge closes a cycle in the conflict graph, representing the absence of opacity or co-opacity.

In addition to applying the graph characterization of CO-Opacity in the STM Haskell and comparing it to the graph characterization of Opacity from TL2, an analysis of



(a) CG from STM Haskell. (b) CG from TL2.

Figure 55 – Conflict graphs resulted from execution of H_1.

what changes could be made to achieve this correctness that STM Haskell does not satisfy. The main difference between the conflict detection in both algorithms boils down to which operations can abort the transactions. STM Haskell is defined as a lazy conflict detection algorithm, meaning that it leaves everything to when a *TryCommit* is called. TL2 looks for conflicts at commit time in the same manner, but also whenever a read operation is called. It was observed that, if this verification of inconsistencies is also applied to a read operation on STM Haskell, it does fundamentally change the algorithm from a lazy conflict detection to an eager one. But, it results in less inconsistent states being generated, therefore more correct histories.

With just a few changes to the production rules of the STM Haskell graph grammar, it is possible achieve this apparent correctness from the executions studied. One way to achieve this is, when a read operation is called, the old values in local buffers are also compared to the shared memory, aborting the transaction if inconsistent. This is the same logic used by TL2, and allows for the Opacity property to include aborted transactions in the history. Another way to achieve this is, when evaluating the history, the flags on the conflict graph node (*Tran*) to be used to change the requirements for rule applications. For Opacity, where *any* transaction could be used to add new conflict relations, CO-Opacity requires them to be flagged as committed (*comm*). This means that when a transaction commits, any aborted transactions and other live transactions with inconsistent reads are not considered for read before write relations.

Looking at history H_1, the second read on T_2, that previously would be executed by STM Haskell, is aborted instead and thus demonstrating Opacity (and CO-Opacity). Adding this verification to a read operation is not enough to make the conflict graphs of both STM Haskell and TL2 to be equal. Mostly for the fact that TL2 starts its verification when a transaction begins (snapshot of the clock), whereas this supposed modified STM Haskell waits for a read operation to do it. But, it is enough to eliminate possible

cycles in the conflict graphs of the histories generated.

6.3 Final Remarks

This chapter described the Graph Grammar (GG) formalism for some study cases of Software Transactional Memory (STM) algorithms. The first algorithm is called CaPR+, that deals with checkpointing and partial rollbacks, making it more complex than standard STM algorithms such as TL2 and SwissTM. The original definition of CaPR+ utilizes data structures to create a local and a global workspace, these workspaces were successfully translated into graph notations by treating them as objects with local information (as vertices with attributes) and pointers to other objects (as edges to other vertices). Similarly as it was described in Chapter 4. All that is left is the logic the algorithm employs on how to manipulate data in these structures, the algorithm defines this in its procedures and that is directly translated into production rules. When translating the procedures to production rules, the biggest complexity issue comes from the various conditional operations (if-else) that each carry a specific set of consequences for the system state. The simplest solution to implement and ease some of that complexity is to split the transactional operation into mutually exclusive production rules. This way, the graph grammar formalization using GROOVE can keep the atomicity level for the execution of the operations (each one is processed in one step), and not have to overcomplicate the pattern match trying to fit every graph transformation the operation requires in one rule. In the traditional graph notation that does not include inheritance, quantifiers and wildcards, the use of multiple steps to process the entire transactional operation is unavoidable. The graph grammar generated from translating the CaPR+ algorithm resulted in a paper publication found in Cardoso; Foss; Du bois (2021).

Another case study presented in this chapter is the graph grammar for the STM library for Haskell library. The main characteristic explored for algorithm is the fact that STM Haskell does not satisfy the opacity correctness criterion, as it allows inconsistent reads to happen. The graph grammar translation is able to simulate STM Haskell's functionality of saving a local buffer for transactional variables accessed and using the buffer to check for conflicts (at commit time). For this case study, an analysis can be made to connect the consequences of the decision making in terms of how and what the transactions are allowed to execute, and how that impacts the correctness of the history being generated. In the case of STM Haskell, if changes could be made to not allow inconsistent reads, the algorithm would possibly generate more correct histories. The graph grammar reflects this via the "reads from" and "read before write" relations in the conflict graph. If changes are made to the production rules to abort read operations before creating inconsistent states, these two relations are the ones that would stop appearing in the conflict graph for that execution. To compare any experimenting

changes made to the STM Haskell graph grammar, a graph grammar formalism for the TL2 algorithm was used, as this algorithm does only generates opaque histories. The histories and conflict graphs generated by an eager conflict detection STM Haskell, although not entirely equivalent to the TL2 counterpart, demonstrated to avoid some of the previous inconsistent states. No further changes were explored in terms of STM algorithms, as it was out of the scope of this thesis. The STM Haskell formalization described here also resulted in a publication that can be found in Cardoso; Foss; Du bois (2022).

7 CONCLUSION

Correctness of Transactional Memory (TM) is an important aspect in the design of TM systems and algorithms. The choice of criteria used may present different levels of strictness, conflict-opacity is more strict than multi-version conflict opacity, and both are more strict than Opacity itself. Previous work on correctness verification have presented automatic ways to verify the correctness of TM systems. Besides opacity, serializability and strict-serializability are also examples of correctness criteria used in these approaches. Existing frameworks and abstract models that formalize transactional memory systems focus on safety verification and often use the aforementioned correctness criteria. More recently, a tool that can automatic check correctness of transactional data structure was proposed (PETERSON; DECHEV, 2017). This shows that correctness verification of transactional memory is a topic of interest, even more if the process can be automated and include various levels of correctness evaluation.

One thing in common in TM correctness verification approaches is the use of a graph to represent some relation between transactions (FLANAGAN; FREUND; YI, 2008; LITZ; DIAS; CHERITON, 2015; PETERSON; DECHEV, 2017). This relation is usually a dependency of variables used or a representation of order between transactional operations, also called happens-before relation. This graph representation of transactional relations can be explored for correctness verification, and one result of this is a graph characterization of opacity presented by Guerraoui; Kapałka (2010). Using a combination of variable dependency and transactional order, the graph characterization of opacity demonstrates the correctness of a history via a simple test of acyclicity in the graph representing the transactions relations. With the ability of proving a correctness criterion via a graph, it seems natural to use a Graph Grammar (GG) to automate part of the verification process.

The first proof of concept developed for this thesis was a GG that focused on a single history as input to generate a conflict graph and test for cycles. The operations in a transactional memory history can be represented using an atomic way, where each operation represents a single step in the system. Or they can be separated in call and response steps, which was the representation used in this first grammar

developed. However, the atomic representation is used in all later iterations because the TM formalism targeted is only at the read/write level and the operations are not complex enough to need multiple steps. The correctness verification developed for this first iteration is close to the final version used in every subsequent transactional memory related GG. This showed that the correctness verification process is rather simple and any variation in the production rules is tied to optimization of when to create conflict graphs.

With the working prototype of verifying correctness of a single history, the next phase of the graph grammar formalism of TM is to deal with full algorithms that generate histories. This introduced a dynamic way to simulate conflict via a system that actually demonstrates decision making and how each action to the shared memory impacts the correctness of the system. A more generic set or production rules were the focus of the study on formalizing an algorithm, these rules mainly focused on representing which type of versioning and conflict detection are used. The possible types are lazy or eager for both versioning and conflict detection. It was observed that in terms of complexity of the methodology, it is simpler to deal with a lazy versioning algorithm rather than an eager one, mainly because rollbacks may require more complex logic as to what to store and how/when to perform the rollback. For conflict detection, both eager and lazy approaches are rather simple to deal with because the complexity comes with resolving the conflict, and not necessarily detecting it. This resolutions often is part of the commit or abort operation.

Using the more generic methodology of a GG for a TM algorithm as basis, some case studies were developed to further understand the graph formalism's capabilities in terms of dealing with complexity and the actual correctness verification. The first one is the CaPR+ algorithm, which excels in the fact that allows partial rollbacks for transactions, instead of restarting the transaction in its entirety. When translating that feature into production rules an approach of mutually exclusive rules was necessary. Instead of using a single rule to enact an entire transactional operation, multiple production rules act as different scenarios of the system's state. The same logic happens in the generic GG, but for the CaPR+ algorithm some rules are more complex than others, most notably anything related to the checkpoint and rollback feature. The graph representation using the tool GROOVE provided a major assistance in compressing multiple actions in each production rule, the downside of using it is sacrificing readability and the complexity of actually creating the production rule.

For the second case study, the STM Haskell library was chosen to create a GG and evaluate its correctness. Differently from the CaPR+ algorithm, the STM Haskell library does not satisfy the correctness criterion opacity. So a GG formalism was used to investigate how this behavior is reflected in the graph characterization of a correctness criterion. In the default STM Haskell algorithm, cycles in the conflict graph are

inevitable, but it was observed that the lazy nature of conflict detection of the STM Haskell library is what produces these cycles. Using a grammar for the TL2 algorithm, which has been proven to be opaque, it was possible to compare the difference in decision making of when to abort in order to avoid inconsistencies. For this case study, making changes to the GG to accommodate a more strict conflict detection resulted in executions, by the modified STM Haskell, that avoid inconsistent states. However, modifying the decision making progress may be considered a fundamental change on the STM Haskell algorithm that would result in an entirely new version being introduced, and that is out of the scope of this thesis and was not pursued any further.

One of the main obstacles found in the methodology of formalizing a STM algorithm using GG is the input of transactions that will generate histories. Not only because an input has to set which operations the system can execute dictating what type of histories will be generated (balance between reads and writes may result in different cases of inconsistencies), but also because too many transactions or too many operations will make the state space too large. This is mainly a concern when using the tool GROOVE that requires a full state space exploration before testing a graph condition, in this case acyclicity. A promising alternative found for this is the use of an event-B model to check for correctness. The theorem prover tool for event-B does not include a visual component to represent states, so it is expected to be capable of dealing with heavier loads than GROOVE.

The event-B model for graph transformation developed for this thesis is based on the first iteration of transactional memory graph grammar. It uses a single history as input and evaluates the acyclicity of its conflict graph. The main difference between using GROOVE and event-B is that event-B does not allow the same complex features in the production rules, mainly quantifier operators (for all, exists, etc). This means that for any operation that could deal with multiple vertices or edges in a single step, now has to use 3 or more. This changes the structure of the grammar in the sense that there are more production rules to execute, and some of them lock the system to a fixed chain of events (if a begin operation starts, nothing else can execute before it finishes). Ultimately the event-B model has the same expression capabilities to emulate the transactional memory operations, it just needs more steps to get there. One benefit of using an event-B model is the capabilities that invariants as state properties bring. The acyclicity property is set up using invariants and they work in the same way as graph conditions in GROOVE, but in event-B each production rule is required to not violate the property for the theorem prover. Other properties for logical and structural correctness can be expressed using invariants, these can be seen as improvements for the type graph or just characteristics of the system as it evolves when production rules are applied.

In conclusion, a graph transformation approach can be used to demonstrate cor-

rectness verification of transactional memory. The methodology can applied directly to TM algorithms which may be useful for new features being introduced to TM that need assistance with proof of correctness.

With the results of the development of this thesis, the following work is highlighted:

- CARDOSO, D. J.; FOSS, L.; DU BOIS, A. R. A Graph Transformation System formalism for Software Transactional Memory Opacity. In: XXIII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2019. p.3–10.
  - In this work, the first prototype of the graph transformation approach was developed. Using GROOVE, this prototype was capable of processing one history at a time and extract its conflict graph to test for acyclicity.
- CARDOSO, D. J.; FOSS, L.; DU BOIS, A. R. A Methodology for Opacity verification for Transactional Memory algorithms using Graph Transformation System. In: VI WORKSHOP-ESCOLA DE INFORMÁTICA TEÓRICA, 2021. p.9–16.
  - The methodology for translating a transactional memory algorithm into graph notations was developed for this work, and this can be seen in Chapter 4 of this thesis.
- CARDOSO, D. J.; FOSS, L.; DU BOIS, A. R. A Graph Transformation System formalism for correctness of Transactional Memory algorithms. In: XXV BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2021. p.49–57.
  - This work presents the results of the methodology being applied to the CaPR+ algorithm as a case study of a complex TM algorithm, as described in Chapter 6.1.
- CARDOSO, D. J.; FOSS, L.; DU BOIS, A. R. Exploring Opacity of Software Transactional Memory in Haskell through Graph Transformation. In: XXVI BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2022. p.15–23.
  - In this paper the methodology is further improved and a new graph characterization is added, the CO-Opacity correctness criterion. The STM Haskell library and the TL2 algorithm are used as a case study to show how decision making by the algorithm influences the outcome of correctness, as described in Chapter 6.2.

For future work, the topic of optimization of the translation step is important to keep in mind, seeing as it is still a fully manual step, any level of automation would be an improvement. The choice of correctness criteria for this thesis focused on opacity and its variations, exploring any new or different correctness criterion is also an aim for

future work. In terms of using GROOVE or event-B to apply the correctness criterion, it would be interesting to explore more in depth both tools to really compare pros and cons. An extension for the event-B model to also incorporate the TM algorithm is another topic to explore. A full implementation of the formalization methodology using event-B requires some work on the production rules, as many features of GROOVE are not directly translated (inheritance, quantifiers, wildcards, etc). With the possibility of support for various graph characterizations of correctness, a tool that uses this formalization and tests for more than one and possibly demonstrates what level of strictness the algorithm in question allows is also something to strive for.

REFERENCES

ABRIAL, J.-R. **Modeling in Event-B**: system and software engineering. [S.l.]: Cambridge University Press, 2010.

ABRIAL, J.-R.; HALLERSTEDE, S. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. **Fundamenta Informaticae**, [S.l.], v.77, n.1-2, p.1–28, 2007.

ABRIAL, J.-R.; HOARE, A. **The B-book**: assigning programs to meanings. [S.l.]: Cambridge university press Cambridge, 1996. v.1.

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. K.; PERI, S. Opacity Proof for CaPR+ Algorithm. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING, 17., 2016, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2016. (ICDCN '16).

ANJANA, P. S. et al. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.83–92.

BACK, R.; SERE, K. Stepwise refinement of action systems. In: INTERNATIONAL CONFERENCE ON MATHEMATICS OF PROGRAM CONSTRUCTION, 1989. **Anais...** [S.l.: s.n.], 1989. p.115–138.

BACKHOUSE, R.; WOUDE, J. van der. Demonic operators and monotype factors. **Mathematical Structures in Computer Science**, [S.l.], v.3, n.4, p.417–433, 1993.

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. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).

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.

BELWAL, C.; CHENG, A. M. K. Lazy Versus Eager Conflict Detection in Software Transactional Memory: A Real-Time Schedulability Perspective. **IEEE Embedded Systems Letters**, [S.l.], v.3, n.1, p.37–41, 2011.

BERGHAMMER, R.; SCHMIDT, G. Relational specifications. **Algebraic Logic**, [S.l.], v.28, p.1993, 1993.

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. **J. ACM**, New York, NY, USA, v.66, n.1, dec 2018.

CARDOSO, D.; FOSS, L.; DU BOIS, A. A Graph Transformation System Formalism for Correctness of Transactional Memory Algorithms. In: BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 25., 2021, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. p.49–57. (SBLP '21).

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, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2019. p.3–10. (SBLP '19).

CARDOSO, D. J.; FOSS, L.; BOIS, A. R. D. A Methodology for Opacity verification for Transactional Memory algorithms using Graph Transformation System. In: VI WORKSHOP-ESCOLA DE INFORMáTICA TEóRICA, 2021, Porto Alegre, RS, Brasil. **Anais...** SBC, 2021. p.9–16.

CARDOSO, D. J.; FOSS, L.; DU BOIS, A. R. Exploring Opacity Software Transactional Memory in Haskell through Graph Transformation. In: XXVI BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2022, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p.15–23. (SBLP '22).

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.8, n.2, p.244–263, apr 1986.

CLEMENTS, A. T. et al. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. **Commun. ACM**, New York, NY, USA, v.60, n.8, p.83–90, jul 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.

COSTA CAVALHEIRO, S. A. da; FOSS, L.; RIBEIRO, L. Theorem proving graph grammars with attributes and negative application conditions. **Theoretical Computer Science**, [S.l.], v.686, p.25–77, 2017.

COURCELLE, B. The expression of graph properties and graph transformations in monadic second-order logic. In: **Handbook Of Graph Grammars And Computing By Graph Transformation**: Volume 1: Foundations. [S.l.]: World Scientific, 1997. p.313–400.

da Costa, S. A.; RIBEIRO, L. Formal Verification of Graph Grammars using Mathematical Induction. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.240, p.43–60, 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

da Costa, S. A.; RIBEIRO, L. Verification of graph grammars using a logical approach. **Science of Computer Programming**, [S.l.], v.77, n.4, p.480–504, 2012. Brazilian Symposium on Formal Methods (SBMF 2008).

DAMRON, P. et al. Hybrid Transactional Memory. **SIGPLAN Notices**, [S.l.], v.41, n.11, p.336–346, 2006.

DEPLOY, E. U. I. P.; RODIN. **Event-B and the Rodin platform**. Disponível em: <http://www.event-b.org/>.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: DISTRIBUTED COMPUTING, 2006, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2006. p.194–208.

DICKERSON, T.; GAZZILLO, P.; HERLIHY, M.; KOSKINEN, E. Adding Concurrency to Smart Contracts. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p.303–312. (PODC '17).

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. Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 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.

DRAGOJEVIć, A.; GUERRAOUI, R.; KAPALKA, M. Stretching Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.44, n.6, p.155–165, jun 2009.

DZIUMA, D.; FATOUROU, P.; KANELLOU, E. Consistency for Transactional Memory Computing. In: GUERRAOUI, R.; ROMANO, P. (Ed.). **Transactional Memory. Foundations, Algorithms, Tools, and Applications**: COST Action Euro-TM IC1001. Cham: Springer International Publishing, 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.

EMERSON, E.; CLARKE, E. M. Using branching time temporal logic to synthesize synchronization skeletons. **Science of Computer Programming**, [S.l.], v.2, n.3, p.241–266, 1982.

EMERSON, E.; HALPERN, J. On branching versus linear time temporal logic. **Journal of the ACM**, [S.l.], v.33, n.1, p.151–178, 1986.

EMMI, M.; MAJUMDAR, R.; MANEVICH, R. Parameterized Verification of Transactional Memories. **SIGPLAN Not.**, New York, NY, USA, v.45, n.6, p.134–145, jun 2010.

FLANAGAN, C.; FREUND, S. N.; YI, J. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. **SIGPLAN Not.**, New York, NY, USA, v.43, n.6, p.293–303, jun 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, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2008. p.175–184. (PPoPP '08).

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. **Proceedings...** Association for Computing Machinery, 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. **SIGARCH Computer Architecture News**, [S.l.], v.21, n.2, p.289–300, 1993.

HIRVE, S.; PALMIERI, R.; RAVINDRAN, B. HiperTM: High performance, fault-tolerant transactional memory. **Theoretical Computer Science**, [S.l.], v.688, p.86–102, 2017. Distributed Computing and Networking.

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. Structural Information and Communication Complexity – SIROCCO 2009.

KHYZHA, A.; ATTIYA, H.; GOTSMAN, A.; RINETZKY, N. Safe Privatization in Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.53, n.1, p.233–245, feb 2018.

KUMAR, P.; PERI, S. Multiversion Conflict Notion for Transactional Memory Systems. **CoRR**, [S.l.], v.abs/1509.04048, 2015.

KUMAR, P.; PERI, S.; VIDYASANKAR, K. A TimeStamp Based Multi-version STM Algorithm. In: DISTRIBUTED COMPUTING AND NETWORKING, 2014, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2014. p.212–226.

KUMARI, S.; PERI, S. **Exploring Progress Guarantees in Multi-Version Software Transactional Memory Systems**. 2019. Tese (Doutorado em Ciência da Computação) — Indian Institute of Technology Hyderabad.

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

LESANI, M.; PALSBERG, J. Decomposing Opacity. In: DISTRIBUTED COMPUTING, 2014, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2014. p.391–405.

LITZ, H.; DIAS, R. J.; CHERITON, D. R. Efficient Correction of Anomalies in Snapshot Isolation Transactions. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.11, n.4, jan 2015.

LITZ, H. et al. SI-TM: Reducing Transactional Memory Abort Rates through Snapshot Isolation. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 19., 2014, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2014. p.383–398. (ASPLOS '14).

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

MANOVIT, C. et al. Testing Implementations of Transactional Memory. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 15., 2006, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2006. p.134–143. (PACT '06).

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

MARLOW, S.; PEYTON JONES, S.; SINGH, S. Runtime Support for Multicore Haskell. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, 14., 2009. **Anais...** Association for Computing Machinery, 2009. p.65–78. (ICFP '09).

MATVEEV, A.; SHAVIT, N. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. **SIGPLAN Not.**, New York, NY, USA, v.50, n.4, p.59–71, mar 2015.

PANKRATIUS, V.; ADL-TABATABAI, A.-R. A Study of Transactional Memory vs. Locks in Practice. In: TWENTY-THIRD ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2011. **Anais...** Association for Computing Machinery, 2011. p.43–52. (SPAA '11).

PAPADIMITRIOU, C. H. The Serializability of Concurrent Database Updates. **J. ACM**, New York, NY, USA, v.26, n.4, p.631–653, oct 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, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2015. p.217–226. (PODC '15).

PETERSON, C.; DECHEV, D. A Transactional Correctness Tool for Abstract Data Types. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.14, n.4, nov 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/>.

RIBEIRO, L.; DOTTI, F. L.; COSTA, S. A. da; DILLENBURG, F. C. Towards theorem proving graph grammars using Event-B. **Electronic Communications of the EASST**, [S.l.], v.30, 2010.

GUERRAOUI, R.; ROMANO, P. (Ed.). **Safety and Deferred Update in Transactional Memory**. Cham: Springer International Publishing, 2015. p.50–71.

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 Transactional Memory. In: STABILIZATION, SAFETY, AND SECURITY OF DISTRIBUTED SYSTEMS, 2010. **Anais...** Springer, 2010. p.388–404.

WEIKUM, G.; VOSSEN, G. **Transactional information systems**: theory, algorithms, and the practice of concurrency control and recovery. [S.l.]: Elsevier, 2001.

ZENG, J. **Augmenting Transactional Memory with the Future Abstraction**. 2020. Tese (Doutorado em Ciência da Computação) — KTH Royal Institute of Technology.

Apendices

APPENDIX A – Software Transactional Memory algorithms

A.1 Checkpointing and Partial Rollback

Checkpointing and Partial Rollback algorithm (CaPR+)

```
1:  procedure ReadTx(t, o, pc)
2:      if o is in t's shared object store then
3:          str.val ← o.val from SOS
4:          return l ← 1(Success);
5:      else if o is in shared memory then
6:          if t.status_flag = RED then
7:              PL = Partially_Rollback(t);
8:              update str.PL = PL
9:              return l ← 0(Rollback);
10:         end if
11:         create checkpoint entry in checkpoint log for o;
12:         str.val ← o.val from Shared Memory
13:         add t to o's readers' list
14:         add o into SOS and set its read flag to 1;
15:         return l ← 1(Success);
16:     else
17:         return l ← 2(Error);
18:     end if
19: end procedure
20: procedure WriteTx(o,t)
21:     if o is a shared object then
22:         if o is in shared object store then
23:             update o in SOS and set its write flag to 1;
24:         else
25:             insert o in SOS and set its write flag to 1;
26:         end if
27:     end if
28: end procedure
29: procedure CommitTx(t)
30:     Assign t's write-set, t.WS = { o | o is in SOS and o's write flag = 1 }
31:     Initialize A = {t};
```

```
32:     for each object o in the t.WS
33:         A = A ∪ active readers of o;
34:     end for
35:     if t.status_flag = RED then
36:         PL = Partially_Rollback(t);
37:         return PL;
38:     end if
39:     for each object wo in t's write-set, t.WS
40:         update wo.value in SM from the local copy of wo;
41:         for each transaction rt in wo's active reader's list
42:             add the objects in t.WS to transaction rt's conflict objects' list;
43:             set transaction rt's status flag to RED;
44:         end for
45:     end for
46:     delete t form actrans;
47:     for each object ro in t's readers-list
48:         delete t from ro's active readers list;
49:     end for
50:     return 0;
51: end procedure
52: procedure Partially_Rollback(t)
53:     identify safest checkpoint - earliest conflicting object;
54:     apply selected checkpoint;
55:     delete t from active reader's list of all object rolled back;
56:     reset status flag to GREEN;
57:     return PL(the new program location);
58: end procedure
```

A.2 Software Transactional Memory Haskell library

Software Transactional Memory Haskell algorithm (STM Haskell)

```
1:  procedure Read(o)
2:      if o is in local store then
3:          return value of local var(o);
4:      else if o is in shared memory then
5:          create local copy of o;
6:          local var(o)'s new value ← value of SM.TVar(o);
7:          local var(o)'s old value ← value of SM.TVar(o);
8:          return value of local var(o);
9:      else
10:         return Error;
```

```
11:     end if
12: end procedure
13: procedure Write(o,v)
14:     if o is in local store then
15:         write v in local var(o);
16:         return Success;
17:     else if o is in shared memory then
18:         create local copy of o;
19:         local var(o)'s new value ← v;
20:         local var(o)'s old value ← v;
21:         return Success;
22:     else
23:         return Error;
24:     end if
25: end procedure
26: procedure Commit()
27:     for each local var x do
28:         if x.old value ≠ value of SM.TVar(x) then
29:             add x to transaction's watchlist;
30:             transaction is flagged for retry;
31:         end if
32:     end for
33:         obtain all locks from write set;
34:     for each local var x do
35:         writes x.new value to SM.TVar(x);
36:     end for
37:         release all locks from write set;
38: end procedure
39: procedure Retry()
40:     if any variable in watchlist receives a change in SM then
41:         erase all local store
42:         restarts transaction
43:     end if
44: end procedure
```

A.3 Transactional Locking 2

Transactional Locking 2 algorithm (TL2)

```
1:  procedure Read(o)
2:      if transaction's read stamp > SM(o)'s write stamp then
3:          erase read set and write set;
```

```
4:              transaction aborts and restarts;
5:          else if o is in write set then
6:              return value of WS(o);
7:          else if o is in read set then
8:              RS(o) ← SM.TVar(o);
9:              return value of RS(o);
10:         else
11:             create new RS(o);
12:             RS(o) ← SM.TVar(o);
13:             return value of RS(o);
14:         end if
15: end procedure
16: procedure Write(o,v)
17:         if o is in write set then
18:             return Success;
19:         else
20:             create new WS(o);
21:             WS(o) ← v;
22:             return Success;
23:         end if
24: end procedure
25: procedure Commit()
26:         for each o in transaction's read set do
27:             if transaction's read stamp > SM(o)'s write stamp then
28:                 erase read set and write set;
29:                 transaction aborts and restarts;
30:             end if
31:         end for
32:         global clock ← global clock + 2;
33:         obtain all locks from write set
34:         for each o in transaction's write set do
35:             SM(o).write stamp ← global clock;
36:             SM(o).value ← WS(o).value;
37:         end for
38:         obtain all locks from write set
39: end procedure
```

APPENDIX B – Translation of STM to GG

B.1 Begin operation

A *begin* operation is divided in 5 total production rules, each one represents an internal step the operation needs. These steps represent a combination of: locking the system, to only allow this operation to execute; dealing with any "loop" necessary, transforming multiple nodes one at a time; and any unlocking needed to transition between steps.

B.1.1 BeginLock rule

Locks the history into a *begin* operation (nothing else can execute), and creates a *CGNode* for the corresponding new transaction.

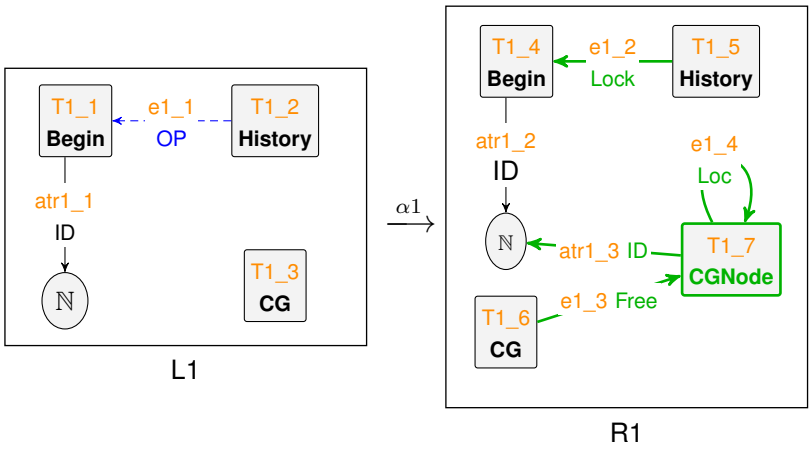


Figure 56 – First step of *Begin* rule.

B.1.2 BeginLoop1 rule

Loops through any **free** and **finished** transaction that does not have a real-time relation (*RT*-edge) with the new transaction and creates the new edge signifying the relation.

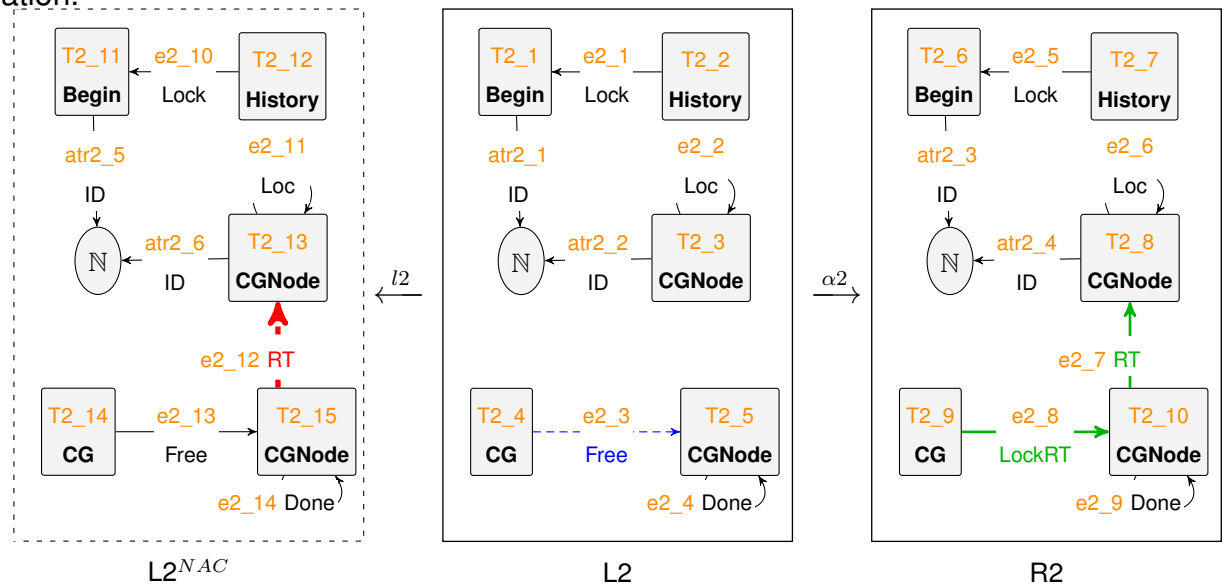


Figure 57 – Second step of *Begin* rule.

B.1.3 BeginLoop1_Release rule

When there are no more transactions to mark with real-time relations, marks the history for the second loop. This prepares to release of the execution for other transactions.

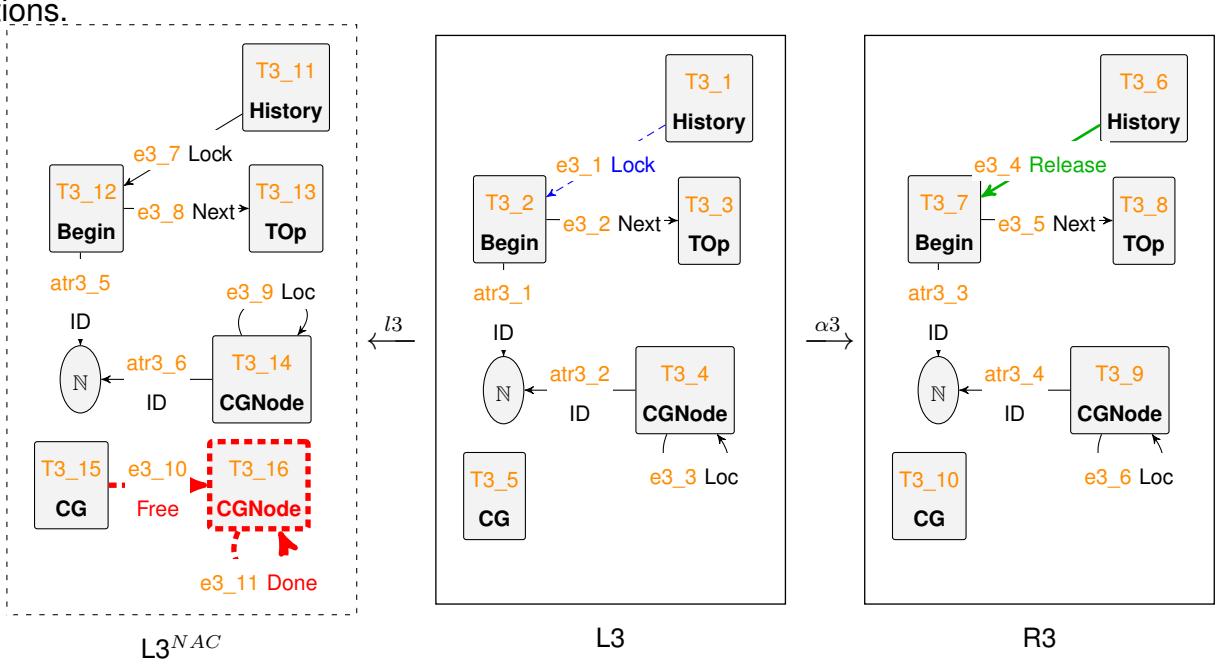


Figure 58 – Third step of *Begin* rule.

B.1.4 BeginLoop2 rule

Loops through any locked *CGNodes* that had a new edge added, and frees them.

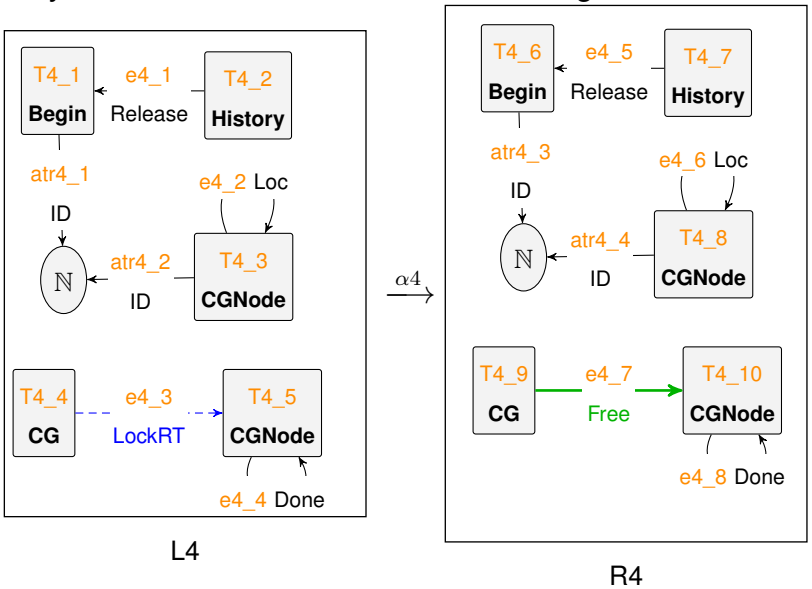


Figure 59 – Fourth step of *Begin* rule.

B.1.5 BeginLoop2_Release rule

When there are no more *CGNodes* to free (or none to begin with), releases the lock on the history and move the *OP*-edge to resume the execution of other operations.

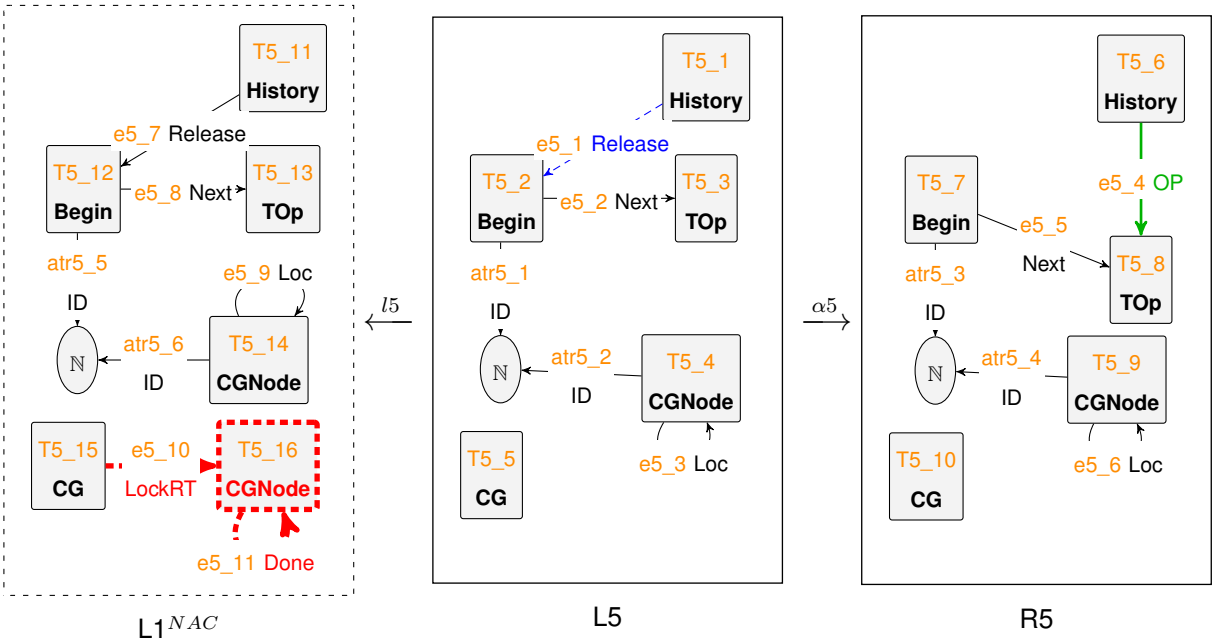


Figure 60 – Fifth and last step of *Begin* rule.

B.2 Commit operation

In the same way as the *begin* operation, a *commit* operation has 5 steps divided into separate production rules. The main goal is to mark any *WW* and *RW*-edges based on the other transactions' *CGNodes*.

B.2.1 CommitLock rule

Locks the history into a *commit* operation, no other operation can execute.

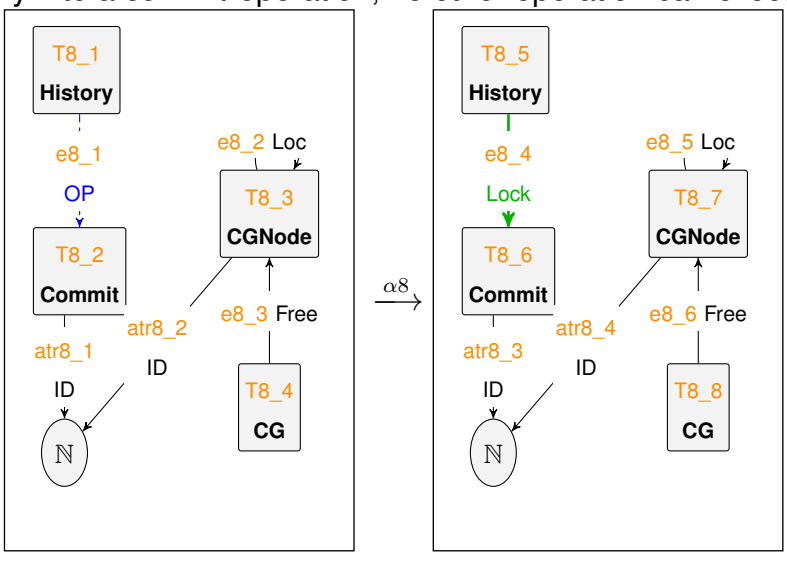
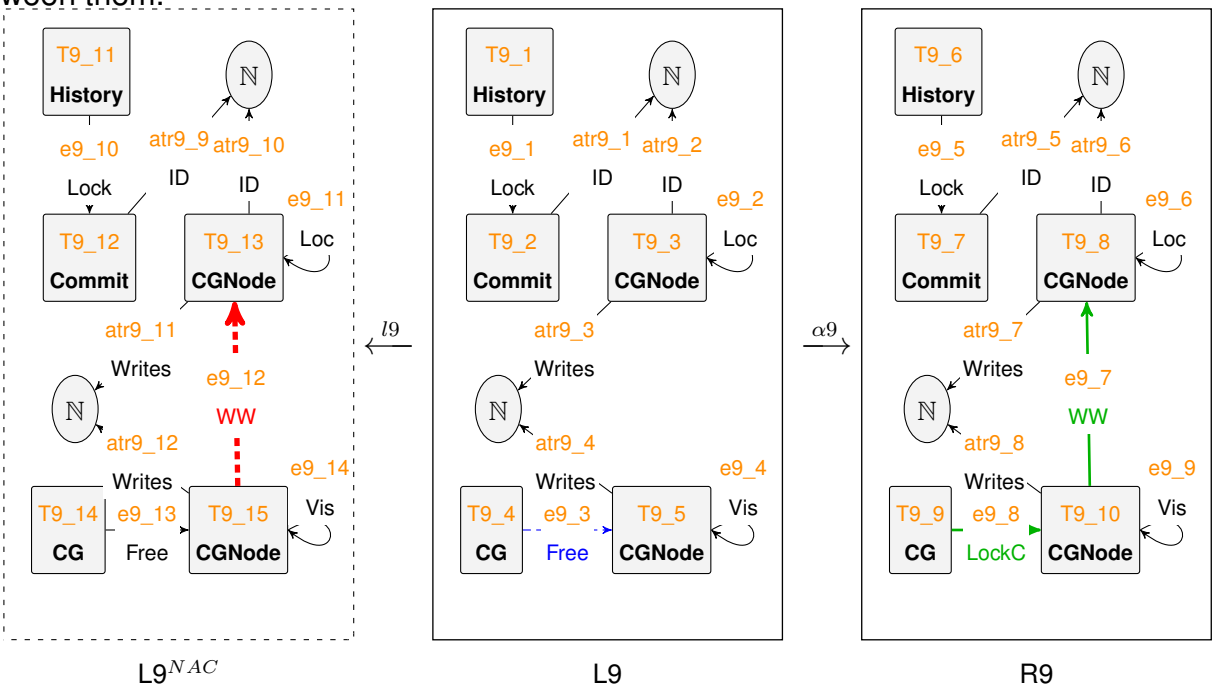


Figure 61 – First step of *Commit* rule.

B.2.2 CommitLoop1 rule

Loops through visible transactions that wrote to the same variables as the transaction executing the commit. Add the new edge *WW*, denoting *write-before-write* relation between them.

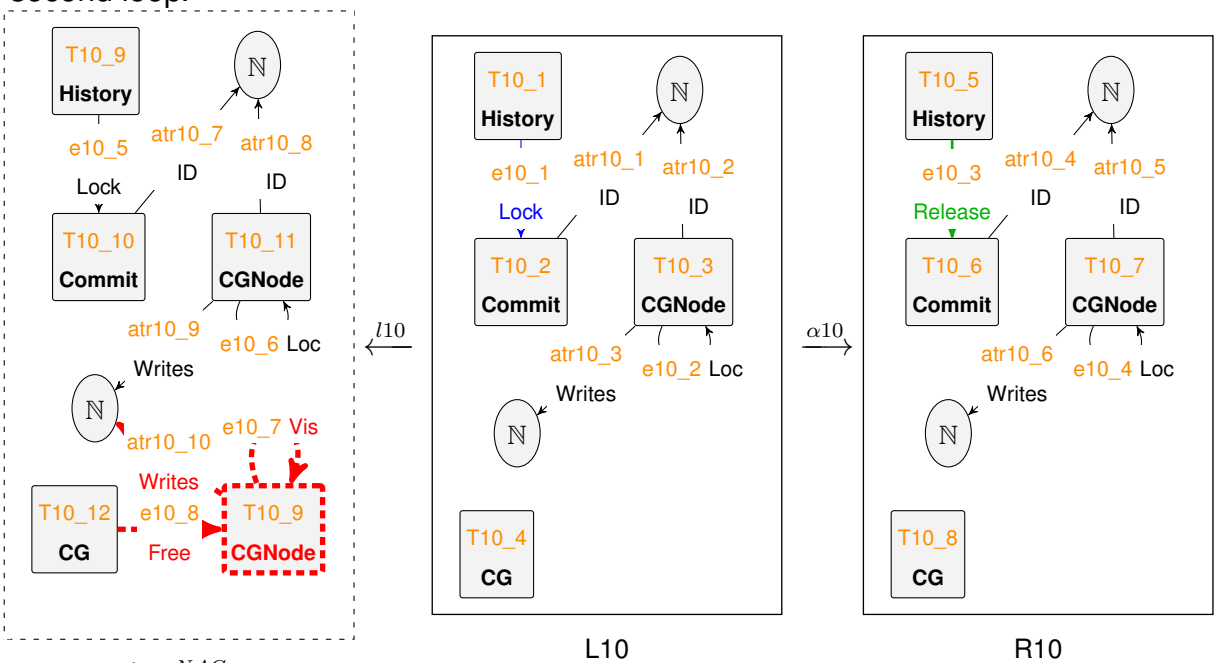


$L9^{NAC}$ L9 R9

Figure 62 – Second step of *Commit* rule.

B.2.3 CommitLoop1_Release rule

As soon as there are no more *WW*-relations to be added, prepare the operation for the second loop.



$L10^{NAC}$ L10 R10

Figure 63 – Third step of *Commit* rule.

B.2.4 CommitLoop2 rule

Loop through any transaction that read a value overwritten by the main transaction (has a *WW*-relation). Adds the *RW*-edge between the transaction that read the variable and the main one. This denotes the *read-before-write* relation, where a value was read and overwritten afterwards, making it inconsistent.

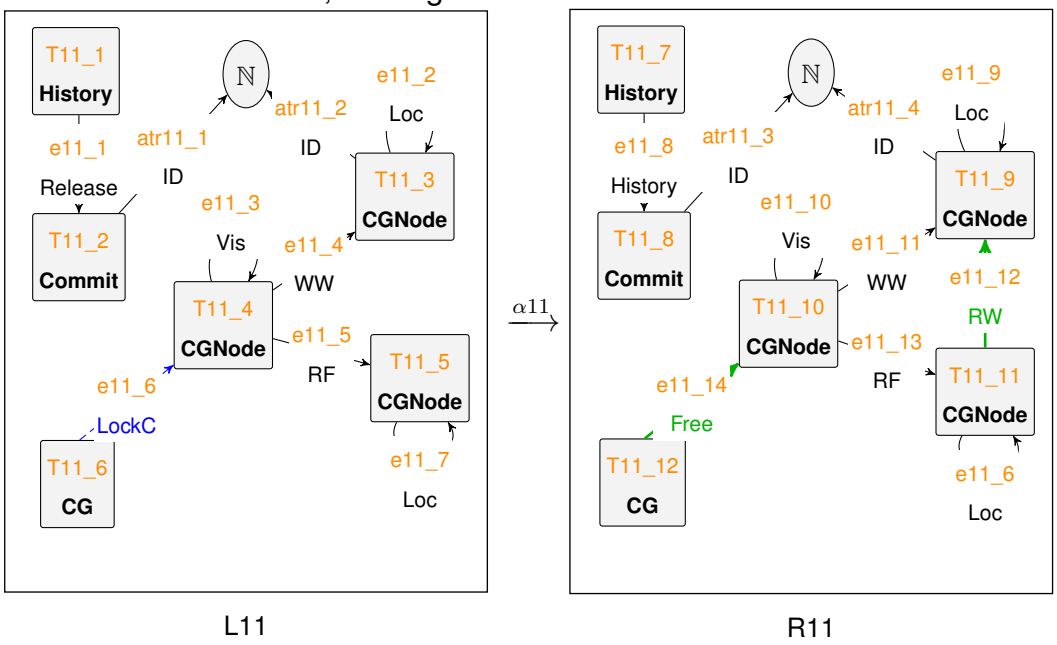


Figure 64 – Fourth step of *Commit* rule.

B.2.5 CommitLoop2_Release rule

Lastly, if there are no more *RW*-edges to add, finishes the execution and unlocks the history for other operations to execute.

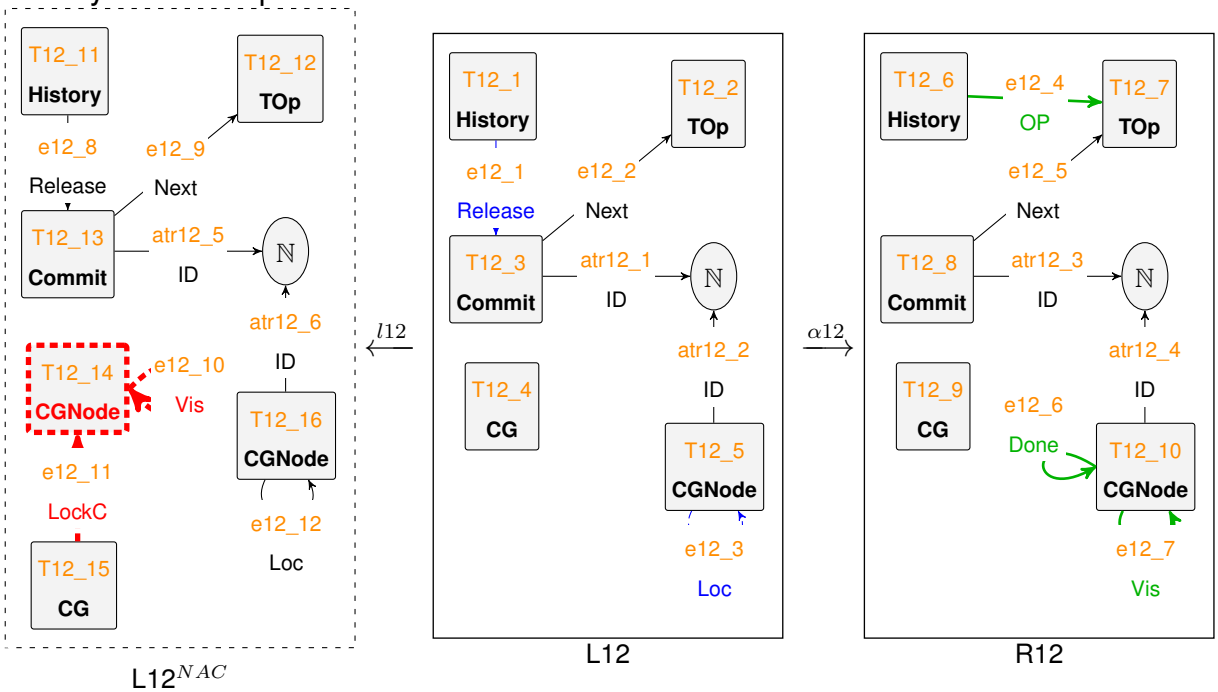


Figure 65 – Fifth and last step of *Commit* rule.

APPENDIX C – Translation of GG to Event-B

C.1 Begin operation

C.1.1 BeginLock rule

See Fig. 56 for graph notation.

sets
  vertL1
  edgeL1
  attrL1
constants
  T1_1 T1_2 T1_3
  e1_1
  atr1_1
  sourceL1
  targetL1
  attrNL1
  attrVL1
  tL1_V
  tL1_E
  tL1_A
axioms
  @axm_vertL1 partition(vertL1, {T1_1}, {T1_2}, {T1_3})
  @axm_edgeL1 partition(edgeL1, {e1_1})
  @axm_attrL1 partition(attrL1, {atr1_1})
  @axm_sourceL1_type sourceL1 ∈ edgeL1 → vertL1
  @axm_sourceL1_def partition(sourceL1, {e1_1 ↦ T1_2})
  @axm_targetL1_type targetL1 ∈ edgeL1 → vertL1
  @axm_targetL1_def partition(targetL1, {e1_1 ↦ T1_1})
  @axm_attrNL1_type attrNL1 ∈ attrL1 → vertL1
  @axm_attrNL1_def partition(attrNL1, {atr1_1 ↦ T1_1})
  @axm_attrVL1_type attrVL1 ∈ attrL1 → ℙ(ℕ)
  @axm_attrVL1_def partition(attrVL1, {atr1_1 ↦ ℕ})
  @axm_tL1_V tL1_V ∈ vertL1 → vertT
  @axm_tL1_V_def partition(tL1_V, {T1_1 ↦ Begin}, {T1_2 ↦ History}, {T1_3 ↦ CG})
  @axm_tL1_E tL1_E ∈ edgeL1 → edgeT
  @axm_tL1_E_def partition(tL1_E, {e1_1 ↦ OP})
  @axm_tL1_A tL1_A ∈ attrL1 → attrT
  @axm_tL1_A_def partition(tL1_A, {atr1_1 ↦ ID})

event BeginLock
any
  mV
  mE
  mA
  T1_7
  e1_2
  e1_3
  e1_4
  atr1_3
  delE
where
  @grd_mV mV ∈ vertL1 → vertG
  @grd_mE mE ∈ edgeL1 → edgeG
  @grd_mA mA ∈ attrL1 → attrG
  @grd_delE delE = [{e1_1}]
  @grd_newT1_7 T1_7 ∈ ℕ\vertG
  @grd_newe1_2 e1_2 ∈ ℕ\edgeG

@grd_newe1_3 e1_3 ∈ ℕ\edgeG
@grd_newe1_4 e1_4 ∈ ℕ\edgeG
@grd_newatr1_3 atr1_3 ∈ ℕ\attrG
@grd_e1_2_e1_3 e1_2 ≠ e1_3
@grd_e1_2_e1_4 e1_2 ≠ e1_4
@grd_e1_3_e1_4 e1_3 ≠ e1_4
@grd_vertices ∀v · v ∈ vertL1 ⇒ tL1_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL1 ⇒ tL1_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL1 ⇒ tL1_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL1 ⇒ mV(sourceL1(e)) = sourceG(mE(e)) ∧ mV(targetL1(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL1 ⇒ mV(attrNL1(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
@act_vertG vertG := vertG ∪ {T1_7}
@act_edgeG edgeG := (edgeG \ delE) ∪ {e1_2, e1_3, e1_4}
@act_attrG attrG := attrG ∪ {atr1_3}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e1_2 ↦ mV(T1_2), e1_3 ↦ mV(T1_3), e1_4 ↦ T1_7}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e1_2 ↦ mV(T1_1), e1_3 ↦ T1_7, e1_4 ↦ T1_7}
@act_attrNG attrNG := attrNG ∪ {atr1_3 ↦ T1_7}
@act_attrVG attrVG := attrVG ∪ {atr1_3 ↦ attrVG(mA(atr1_1))}
@act_tG_V tG_V := tG_V ∪ {T1_7 ↦ CGNode}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e1_2 ↦ Lock, e1_3 ↦ Free, e1_4 ↦ Loc}
@act_tG_A tG_A := tG_A ∪ {atr1_3 ↦ ID}
end

C.1.2 BeginLoop1 rule

See Fig. 57 for graph notation.

sets
vertL2
edgeL2
attrL2
constants
T2_1 T2_2 T2_3 T2_4 T2_5
e2_1 e2_2 e2_3 e2_4
atr2_1 atr2_2
sourceL2
targetL2
attrNL2
attrVL2
tL2_V
tL2_E
tL2_A
axioms
@axm_vertL2 partition(vertL2, {T2_1}, {T2_2}, {T2_3}, {T2_4}, {T2_5})
@axm_edgeL2 partition(edgeL2, {e2_1}, {e2_2}, {e2_3}, {e2_4})
@axm_attrL2 partition(attrL2, {atr2_1}, {atr2_2})
@axm_sourceL2_type sourceL2 ∈ edgeL2 → vertL2
@axm_sourceL2_def partition(sourceL2, {e2_1 ↦ T2_2}, {e2_2 ↦ T2_3}, {e2_3 ↦ T2_4}, {e2_4 ↦ T2_5})
@axm_targetL2_type targetL2 ∈ edgeL2 → vertL2
@axm_targetL2_def partition(targetL2, {e2_1 ↦ T2_1}, {e2_2 ↦ T2_3}, {e2_3 ↦ T2_5}, {e2_4 ↦ T2_5})
@axm_attrNL2_type attrNL2 ∈ attrL2 → vertL2
@axm_attrNL2_def partition(attrNL2, {atr2_1 ↦ T2_1}, {atr2_2 ↦ T2_3})
@axm_attrVL2_type attrVL2 ∈ attrL2 → ℙ(ℕ)
@axm_attrVL2_def partition(attrVL2, {atr2_1 ↦ ℕ}, {atr2_2 ↦ ℕ})
@axm_tL2_V tL2_V ∈ vertL2 → vertT
@axm_tL2_V_def partition(tL2_V, {T2_1 ↦ Begin}, {T2_2 ↦ History}, {T2_3 ↦ CGNode}, {T2_4 ↦ CG}, {T2_5 ↦ CGNode})
@axm_tL2_E tL2_E ∈ edgeL2 → edgeT

@axm_tL2_E_def partition(tL2_E, {e2_1 ↦ Lock}, {e2_2 ↦ Loc}, {e2_3 ↦ Free}, {e2_4 ↦ Done})
@axm_tL2_A tL2_A ∈ attrL2 → attrT
@axm_tL2_A_def partition(tL2_A, {atr2_1 ↦ ID}, {atr2_2 ↦ ID})

event BeginLoop1
any
  mV
  mE
  mA
  e2_8
  e2_7
  delE
where
  @grd_mV mV ∈ vertL2 → vertG
  @grd_mE mE ∈ edgeL2 → edgeG
  @grd_mA mA ∈ attrL2 → attrG
  @grd_delE delE = [{e2_3}]
  @grd_newe2_8 e2_8 ∈ ℕ\edgeG
  @grd_newe2_7 e2_7 ∈ ℕ\edgeG
  @grd_e2_8_e2_7 e2_8 ≠ e2_7
  @grd_vertices ∀v · v ∈ vertL2 ⇒ tL2_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL2 ⇒ tL2_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL2 ⇒ tL2_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL2 ⇒ mV(sourceL2(e)) = sourceG(mE(e)) ∧ mV(targetL2(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL2 ⇒ mV(attrNL2(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
  @grd_NAC_E1 ¬(∃forbRT·
                forbRT ⊆ edgeG \ mE[edgeL2] ∧ tG_E(forbRT) = RT ∧
                sourceG(forbRT) = mV(T2_5) ∧ targetG(forbRT) = mV(T2_3))
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e2_8, e2_7}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e2_8 ↦ mV(T2_4), e2_7 ↦ mV(T2_5)}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e2_8 ↦ mV(T2_5), e2_7 ↦ mV(T2_3)}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e2_8 ↦ LockRT, e2_7 ↦ RT}
end

C.1.3 BeginLoop1_Release rule

See Fig. 58 for graph notation.

sets
  vertL3
  edgeL3
  attrL3
constants
  T3_1 T3_2 T3_3 T3_4 T3_5
  e3_1 e3_2 e3_3
  atr3_1 atr3_2
  sourceL3
  targetL3
  attrNL3
  attrVL3
  tL3_V
  tL3_E
  tL3_A
axioms
  @axm_vertL3 partition(vertL3, {T3_1}, {T3_2}, {T3_3}, {T3_4}, {T3_5})
  @axm_edgeL3 partition(edgeL3, {e3_1}, {e3_2}, {e3_3})

@axm_attrL3 partition(attrL3, {atr3_1}, {atr3_2})
@axm_sourceL3_type sourceL3 ∈ edgeL3 → vertL3
@axm_sourceL3_def partition(sourceL3, {e3_1 ↦ T3_1}, {e3_2 ↦ T3_2}, {e3_3 ↦ T3_4})
@axm_targetL3_type targetL3 ∈ edgeL3 → vertL3
@axm_targetL3_def partition(targetL3, {e3_1 ↦ T3_2}, {e3_2 ↦ T3_3}, {e3_3 ↦ T3_4})
@axm_attrNL3_type attrNL3 ∈ attrL3 → vertL3
@axm_attrNL3_def partition(attrNL3, {atr3_1 ↦ T3_2}, {atr3_2 ↦ T3_4})
@axm_attrVL3_type attrVL3 ∈ attrL3 → ℙ(ℕ)
@axm_attrVL3_def partition(attrVL3, {atr3_1 ↦ ℕ}, {atr3_2 ↦ ℕ})
@axm_tL3_V tL3_V ∈ vertL3 → vertT
@axm_tL3_V_def partition(tL3_V, {T3_1 ↦ History}, {T3_2 ↦ Begin}, {T3_3 ↦ Begin}, {T3_4 ↦ CGNode}, {T3_5 ↦ CG})
@axm_tL3_E tL3_E ∈ edgeL3 → edgeT
@axm_tL3_E_def partition(tL3_E, {e3_1 ↦ Lock}, {e3_2 ↦ Next}, {e3_3 ↦ Loc})
@axm_tL3_A tL3_A ∈ attrL3 → attrT
@axm_tL3_A_def partition(tL3_A, {atr3_1 ↦ ID}, {atr3_2 ↦ ID})

event BeginLoop1_Release
any
  mV
  mE
  mA
  e3_4
  delE
where
  @grd_mV mV ∈ vertL3 → vertG
  @grd_mE mE ∈ edgeL3 → edgeG
  @grd_mA mA ∈ attrL3 → attrG
  @grd_delE delE = [{e3_1}]
  @grd_newe3_4 e3_4 ∈ ℕ\edgeG
  @grd_vertices ∀v · v ∈ vertL3 ⇒ tL3_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL3 ⇒ tL3_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL3 ⇒ tL3_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL3 ⇒ mV(sourceL3(e)) = sourceG(mE(e)) ∧ mV(targetL3(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL3 ⇒ mV(attrNL3(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
  @grd_NAC_V1 ¬(∃forbT3_16·
      forbT3_16 ⊆ vertG \ mV[vertL3] ∧ tG_V(forbT3_16) = CGNode ∧
      (∃forbe3_10·
          forbe3_10 ⊆ edgeG \ mE[edgeL3] ∧ tG_E(forbe3_10) = Free ∧
          sourceG(forbe3_10) = mV(mV(T3_5)) ∧ targetG(forbe3_10) = mV(forbT3_16)
      (∃forbe3_11·
          forbe3_11 ⊆ edgeG \ mE[edgeL3] ∧ tG_E(forbe3_11) = Done ∧
          sourceG(forbe3_11) = mV(forbT3_16) ∧ targetG(forbe3_11) = mV(forbT3_16))
then
  @act_edgeG edgeG := (edgeG \ delE) ∪ {e3_4}
  @act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e3_4 ↦ mV(T3_1)}
  @act_targetG targetG := (delE ⩤ targetG) ∪ {e3_4 ↦ mV(T3_2)}
  @act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e3_4 ↦ Release}
end

C.1.4 BeginLoop2 rule

See Fig. 59 for graph notation.

sets
  vertL4
  edgeL4
  attrL4

constants
T4_1 T4_2 T4_3 T4_4 T4_5
e4_1 e4_2 e4_3 e4_4
atr4_1 atr4_2
sourceL4
targetL4
attrNL4
attrVL4
tL4_V
tL4_E
tL4_A
axioms
@axm_vertL4 partition(vertL4, {T4_1}, {T4_2}, {T4_3}, {T4_4}, {T4_5})
@axm_edgeL4 partition(edgeL4, {e4_1}, {e4_2}, {e4_3}, {e4_4})
@axm_attrL4 partition(attrL4, {atr4_1}, {atr4_2})
@axm_sourceL4_type sourceL4 ∈ edgeL4 → vertL4
@axm_sourceL4_def partition(sourceL4, {e4_1 ↦ T4_2}, {e4_2 ↦ T4_3}, {e4_3 ↦ T4_4}, {e4_4 ↦ T4_5})
@axm_targetL4_type targetL4 ∈ edgeL4 → vertL4
@axm_targetL4_def partition(targetL4, {e4_1 ↦ T4_1}, {e4_2 ↦ T4_3}, {e4_3 ↦ T4_5}, {e4_4 ↦ T4_5})
@axm_attrNL4_type attrNL4 ∈ attrL4 → vertL4
@axm_attrNL4_def partition(attrNL4, {atr4_1 ↦ T4_1}, {atr4_2 ↦ T4_3})
@axm_attrVL4_type attrVL4 ∈ attrL4 → ℙ(ℕ)
@axm_attrVL4_def partition(attrVL4, {atr4_1 ↦ ℕ}, {atr4_2 ↦ ℕ})
@axm_tL4_V tL4_V ∈ vertL4 → vertT
@axm_tL4_V_def partition(tL4_V, {T4_1 ↦ Begin}, {T4_2 ↦ History}, {T4_3 ↦ CGNode}, {T4_4 ↦ CG}, {T4_5 ↦ CGNode})
@axm_tL4_E tL4_E ∈ edgeL4 → edgeT
@axm_tL4_E_def partition(tL4_E, {e4_1 ↦ Release}, {e4_2 ↦ Loc}, {e4_3 ↦ LockRT}, {e4_4 ↦ Done})
@axm_tL4_A tL4_A ∈ attrL4 → attrT
@axm_tL4_A_def partition(tL4_A, {atr4_1 ↦ ID}, {atr4_2 ↦ ID})

event BeginLoop2
any
mV
mE
mA
e4_7
delE
where
@grd_mV mV ∈ vertL4 → vertG
@grd_mE mE ∈ edgeL4 → edgeG
@grd_mA mA ∈ attrL4 → attrG
@grd_delE delE = [{e4_3}]
@grd_newe4_7 e4_7 ∈ ℕ\edgeG
@grd_vertices ∀v · v ∈ vertL4 ⇒ tL4_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL4 ⇒ tL4_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL4 ⇒ tL4_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL4 ⇒ mV(sourceL4(e)) = sourceG(mE(e)) ∧ mV(targetL4(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL4 ⇒ mV(attrNL4(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e4_7}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e4_7 ↦ mV(T4_4)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e4_7 ↦ mV(T4_5)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e4_7 ↦ Free}
end

C.1.5 BeginLoop2_Release rule

See Fig. 60 for graph notation.

sets
  vertL5
  edgeL5
  attrL5
constants
  T5_1 T5_2 T5_3 T5_4 T5_5
  e5_1 e5_2 e5_3
  atr5_1 atr5_2
  sourceL5
  targetL5
  attrNL5
  attrVL5
  tL5_V
  tL5_E
  tL5_A
axioms
  @axm_vertL5 partition(vertL5, {T5_1}, {T5_2}, {T5_3}, {T5_4}, {T5_5})
  @axm_edgeL5 partition(edgeL5, {e5_1}, {e5_2}, {e5_3})
  @axm_attrL5 partition(attrL5, {atr5_1}, {atr5_2})
  @axm_sourceL5_type sourceL5 ∈ edgeL5 → vertL5
  @axm_sourceL5_def partition(sourceL5, {e5_1 ↦ T5_1}, {e5_2 ↦ T5_2}, {e5_3 ↦ T5_4})
  @axm_targetL5_type targetL5 ∈ edgeL5 → vertL5
  @axm_targetL5_def partition(targetL5, {e5_1 ↦ T5_2}, {e5_2 ↦ T5_3}, {e5_3 ↦ T5_4})
  @axm_attrNL5_type attrNL5 ∈ attrL5 → vertL5
  @axm_attrNL5_def partition(attrNL5, {atr5_1 ↦ T5_2}, {atr5_2 ↦ T5_4})
  @axm_attrVL5_type attrVL5 ∈ attrL5 → ℙ(ℕ)
  @axm_attrVL5_def partition(attrVL5, {atr5_1 ↦ ℕ}, {atr5_2 ↦ ℕ})
  @axm_tL5_V tL5_V ∈ vertL5 → vertT
  @axm_tL5_V_def partition(tL5_V, {T5_1 ↦ History}, {T5_2 ↦ Begin}, {T5_3 ↦ Begin}, {T5_4 ↦ CGNode}, {T5_5 ↦ CG})
  @axm_tL5_E tL5_E ∈ edgeL5 → edgeT
  @axm_tL5_E_def partition(tL5_E, {e5_1 ↦ Release}, {e5_2 ↦ Next}, {e5_3 ↦ Loc})
  @axm_tL5_A tL5_A ∈ attrL5 → attrT
  @axm_tL5_A_def partition(tL5_A, {atr5_1 ↦ ID}, {atr5_2 ↦ ID})

event BeginLoop2_Release
any
  mV
  mE
  mA
  e5_4
  delE
where
  @grd_mV mV ∈ vertL5 → vertG
  @grd_mE mE ∈ edgeL5 → edgeG
  @grd_mA mA ∈ attrL5 → attrG
  @grd_delE delE = [{e5_1}]
  @grd_newe5_4 e5_4 ∈ ℕ\edgeG
  @grd_vertices ∀v · v ∈ vertL5 ⇒ tL5_V(v) = tG_V(mV(v))
  @grd_edges ∀e · e ∈ edgeL5 ⇒ tL5_E(e) = tG_E(mE(e))
  @grd_attrs ∀a · a ∈ attrL5 ⇒ tL5_A(a) = tG_A(mA(a))
  @grd_srctgt ∀e · e ∈ edgeL5 ⇒ mV(sourceL5(e)) = sourceG(mE(e)) ∧ mV(targetL5(e)) = targetG(mE(e))
  @grd_attrComp ∀a · a ∈ attrL5 ⇒ mV(attrNL5(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
  @grd_NAC_V1 ¬(∃forbT5_16·
      forbT5_16 ⊆ vertG \ mV[vertL5] ∧ tG_V(forbT5_16) = CGNode ∧
      (∃forbe5_10·
          forbe5_10 ⊆ edgeG \ mE[edgeL5] ∧ tG_E(forbe5_10) = LockRT ∧
          sourceG(forbe5_10) = mV(mV(T5_5)) ∧ targetG(forbe5_10) = mV(forbT5_16)
      (∃forbe5_11·

forbe5_11 ⊆ edgeG \ mE[edgeL5] ∧ tG_E(forbe5_11) = Done ∧
sourceG(forbe5_11) = mV(mV(T5_5)) ∧ targetG(forbe5_11) = mV(forbT5_16))

then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e5_4}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e5_4 ↦ mV(T5_1)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e5_4 ↦ mV(T5_3)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e5_4 ↦ OP}
end

C.2 Commit operation

C.2.1 CommitLock rule

See Fig. 61 for graph notation.

sets
vertL8
edgeL8
attrL8
constants
T8_1 T8_2 T8_3 T8_4
e8_1 e8_2 e8_3
atr8_1 atr8_2
sourceL8
targetL8
attrNL8
attrVL8
tL8_V
tL8_E
tL8_A
axioms
@axm_vertL8 partition(vertL8, {T8_1}, {T8_2}, {T8_3}, {T8_4})
@axm_edgeL8 partition(edgeL8, {e8_1}, {e8_2}, {e8_3})
@axm_attrL8 partition(attrL8, {atr8_1}, {atr8_2})
@axm_sourceL8_type sourceL8 ∈ edgeL8 → vertL8
@axm_sourceL8_def partition(sourceL8, {e8_1 ↦ T8_1}, {e8_2 ↦ T8_3}, {e8_3 ↦ T8_4})
@axm_targetL8_type targetL8 ∈ edgeL8 → vertL8
@axm_targetL8_def partition(targetL8, {e8_1 ↦ T8_2}, {e8_2 ↦ T8_3}, {e8_3 ↦ T8_3})
@axm_attrNL8_type attrNL8 ∈ attrL8 → vertL8
@axm_attrNL8_def partition(attrNL8, {atr8_1 ↦ T8_2}, {atr8_2 ↦ T8_3})
@axm_attrVL8_type attrVL8 ∈ attrL8 → ℙ(ℕ)
@axm_attrVL8_def partition(attrVL8, {atr8_1 ↦ ℕ}, {atr8_2 ↦ ℕ})
@axm_tL8_V tL8_V ∈ vertL8 → vertT
@axm_tL8_V_def partition(tL8_V, {T8_1 ↦ History}, {T8_2 ↦ Commit}, {T8_3 ↦ CGNode}, {T8_4 ↦ CG})
@axm_tL8_E tL8_E ∈ edgeL8 → edgeT
@axm_tL8_E_def partition(tL8_E, {e8_1 ↦ OP}, {e8_2 ↦ Loc}, {e8_3 ↦ Free})
@axm_tL8_A tL8_A ∈ attrL8 → attrT
@axm_tL8_A_def partition(tL8_A, {atr8_1 ↦ ID}, {atr8_2 ↦ ID})

event CommitLock
any
mV
mE
mA
e8_4
delE

where
@grd_mV mV ∈ vertL8 → vertG
@grd_mE mE ∈ edgeL8 → edgeG
@grd_mA mA ∈ attrL8 → attrG
@grd_delE delE = [{e8_1}]
@grd_newe8_4 e8_4 ∈ ℕ\edgeG
@grd_vertices ∀v · v ∈ vertL8 ⇒ tL8_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL8 ⇒ tL8_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL8 ⇒ tL8_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL8 ⇒ mV(sourceL8(e)) = sourceG(mE(e)) ∧ mV(targetL8(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL8 ⇒ mV(attrNL8(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e8_4}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e8_4 ↦ mV(T8_1)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e8_4 ↦ mV(T8_2)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e8_4 ↦ Lock}
end

C.2.2 CommitLoop1 rule

See Fig. 62 for graph notation.

sets
vertL9
edgeL9
attrL9
constants
T9_1 T9_2 T9_3 T9_4 T9_5
e9_1 e9_2 e9_3 e9_4
atr9_1 atr9_2 atr9_3 atr9_4
sourceL9
targetL9
attrNL9
attrVL9
tL9_V
tL9_E
tL9_A
axioms
@axm_vertL9 partition(vertL9, {T9_1}, {T9_2}, {T9_3}, {T9_4}, {T9_5})
@axm_edgeL9 partition(edgeL9, {e9_1}, {e9_2}, {e9_3}, {e9_4})
@axm_attrL9 partition(attrL9, {atr9_1}, {atr9_2}, {atr9_3}, {atr9_4})
@axm_sourceL9_type sourceL9 ∈ edgeL9 → vertL9
@axm_sourceL9_def partition(sourceL9, {e9_1 ↦ T9_1}, {e9_2 ↦ T9_3}, {e9_3 ↦ T9_4}, {e9_4 ↦ T9_5})
@axm_targetL9_type targetL9 ∈ edgeL9 → vertL9
@axm_targetL9_def partition(targetL9, {e9_1 ↦ T9_2}, {e9_2 ↦ T9_3}, {e9_3 ↦ T9_5}, {e9_4 ↦ T9_5})
@axm_attrNL9_type attrNL9 ∈ attrL9 → vertL9
@axm_attrNL9_def partition(attrNL9, {atr9_1 ↦ T9_2}, {atr9_2 ↦ T9_3}, {atr9_3 ↦ T9_3}, {atr9_4 ↦ T9_5})
@axm_attrVL9_type attrVL9 ∈ attrL9 → ℙ(ℕ)
@axm_attrVL9_def partition(attrVL9, {atr9_1 ↦ ℕ}, {atr9_2 ↦ ℕ}, {atr9_3 ↦ ℕ}, {atr9_4 ↦ ℕ})
@axm_tL9_V tL9_V ∈ vertL9 → vertT
@axm_tL9_V_def partition(tL9_V, {T9_1 ↦ History}, {T9_2 ↦ Commit}, {T9_3 ↦ CGNode}, {T9_4 ↦ CG}, {T9_5 ↦ CGNode})
@axm_tL9_E tL9_E ∈ edgeL9 → edgeT
@axm_tL9_E_def partition(tL9_E, {e9_1 ↦ Lock}, {e9_2 ↦ Loc}, {e9_3 ↦ Free}, {e9_4 ↦ Vis})
@axm_tL9_A tL9_A ∈ attrL9 → attrT
@axm_tL9_A_def partition(tL9_A, {atr9_1 ↦ ID}, {atr9_2 ↦ ID}, {atr9_3 ↦ Writes}, {atr9_4 ↦ Writes})

event CommitLock_Loop1

any
mV
mE
mA
e9_7
e9_8
delE
where
@grd_mV mV ∈ vertL9 → vertG
@grd_mE mE ∈ edgeL9 → edgeG
@grd_mA mA ∈ attrL9 → attrG
@grd_delE delE = [{e9_3}]
@grd_newe9_7 e9_7 ∈ ℕ\edgeG
@grd_newe9_8 e9_8 ∈ ℕ\edgeG
@grd_e9_7_e9_8 e9_7 ≠ e9_8
@grd_vertices ∀v · v ∈ vertL9 ⇒ tL9_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL9 ⇒ tL9_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL9 ⇒ tL9_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL9 ⇒ mV(sourceL9(e)) = sourceG(mE(e)) ∧ mV(targetL9(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL9 ⇒ mV(attrNL9(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
@grd_NAC_E1 ¬(∃forbWW·
forbWW ⊆ edgeG \ mE[edgeL9] ∧ tG_E(forbWW) = WW ∧
sourceG(forbWW) = mV(T9_5) ∧ targetG(forbWW) = mV(T9_3))
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e9_7, e9_8}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e9_7 ↦ mV(T9_5), e9_8 ↦ mV(T9_4)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e9_7 ↦ mV(T9_3), e9_8 ↦ mV(T9_5)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e9_7 ↦ WW, e9_8 ↦ LockC}
end

C.2.3 CommitLoop1_Release rule

See Fig. 63 for graph notation.

sets
vertL10
edgeL10
attrL10
constants
T10_1 T10_2 T10_3 T10_4
e10_1 e10_2
atr10_1 atr10_2 atr10_3
sourceL10
targetL10
attrNL10
attrVL10
tL10_V
tL10_E
tL10_A
axioms
@axm_vertL10 partition(vertL10, {T10_1}, {T10_2}, {T10_3}, {T10_4})
@axm_edgeL10 partition(edgeL10, {e10_1}, {e10_2})
@axm_attrL10 partition(attrL10, {atr10_1}, {atr10_2}, {atr10_3})
@axm_sourceL10_type sourceL10 ∈ edgeL10 → vertL10
@axm_sourceL10_def partition(sourceL10, {e10_1 ↦ T10_1}, {e10_2 ↦ T10_3})
@axm_targetL10_type targetL10 ∈ edgeL10 → vertL10
@axm_targetL10_def partition(targetL10, {e10_1 ↦ T10_2}, {e10_2 ↦ T10_3})
@axm_attrNL10_type attrNL10 ∈ attrL10 → vertL10

@axm_attrNL10_def partition(attrNL10, {atr10_1 ↦ T10_2}, {atr10_2 ↦ T10_3}, {atr10_3 ↦ T10_3})
@axm_attrVL10_type attrVL10 ∈ attrL10 → ℙ(ℕ)
@axm_attrVL10_def partition(attrVL10, {atr10_1 ↦ ℕ}, {atr10_2 ↦ ℕ}, {atr10_3 ↦ ℕ})
@axm_tL10_V tL10_V ∈ vertL10 → vertT
@axm_tL10_V_def partition(tL10_V, {T10_1 ↦ History}, {T10_2 ↦ Commit}, {T10_3 ↦ CGNode}, {T10_4 ↦ CG})
@axm_tL10_E tL10_E ∈ edgeL10 → edgeT
@axm_tL10_E_def partition(tL10_E, {e10_1 ↦ Lock}, {e10_2 ↦ Loc})
@axm_tL10_A tL10_A ∈ attrL10 → attrT
@axm_tL10_A_def partition(tL10_A, {atr10_1 ↦ ID}, {atr10_2 ↦ ID}, {atr10_3 ↦ Writes})

event CommitLoop1_Release
any
mV
mE
mA
e10_3
delE
where
@grd_mV mV ∈ vertL10 → vertG
@grd_mE mE ∈ edgeL10 → edgeG
@grd_mA mA ∈ attrL10 → attrG
@grd_delE delE = [{e10_1}]
@grd_newe10_3 e10_3 ∈ ℕ\edgeG
@grd_vertices ∀v · v ∈ vertL10 ⇒ tL10_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL10 ⇒ tL10_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL10 ⇒ tL10_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL10 ⇒ mV(sourceL10(e)) = sourceG(mE(e)) ∧ mV(targetL10(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL10 ⇒ mV(attrNL10(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
@grd_NAC_V1 ¬(∃forbT10_9·
forbT10_9 ⊆ vertG \ mV[vertL10] ∧ tG_V(forbT10_9) = CGNode ∧
(∃forbe10_7·
forbe10_7 ⊆ edgeG \ mE[edgeL10] ∧ tG_E(forbe10_7) = Loc ∧
sourceG(forbe10_7) = mV(forbT10_9) ∧ targetG(forbe10_7) = mV(forbT10_9)
(∃forbe10_8·
forbe10_8 ⊆ edgeG \ mE[edgeL10] ∧ tG_E(forbe10_8) = Free ∧
sourceG(forbe10_8) = mV(mV(T10_4)) ∧ targetG(forbe10_8) = mV(forbT10_9))
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e10_3}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e10_3 ↦ mV(T10_1)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e10_3 ↦ mV(T10_2)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e10_3 ↦ Release}
end

C.2.4 CommitLoop2 rule

See Fig. 64 for graph notation.

sets
vertL11
edgeL11
attrL11
constants
T11_1 T11_2 T11_3 T11_4 T11_5 T11_6
e11_1 e11_2 e11_3 e11_4 e11_5 e11_6 e11_7
atr11_1 atr11_2
sourceL11
targetL11

attrNL11
attrVL11
tL11_V
tL11_E
tL11_A
axioms
@axm_vertL11 partition(vertL11, {T11_1}, {T11_2}, {T11_3}, {T11_4}, {T11_5}, {T11_6})
@axm_edgeL11 partition(edgeL11, {e11_1}, {e11_2}, {e11_3}, {e11_4}, {e11_5}, {e11_6}, {e11_7})
@axm_attrL11 partition(attrL11, {atr11_1}, {atr11_2})
@axm_sourceL11_type sourceL11 ∈ edgeL11 → vertL11
@axm_sourceL11_def partition(sourceL11, {e11_1 ↦ T11_1}, {e11_2 ↦ T11_3}, {e11_3 ↦ T11_4}, {e11_4 ↦ T11_4}, {e11_5 ↦ T11_4}, {e11_6 ↦ T11_6}, {e11_7 ↦ T11_5})
@axm_targetL11_type targetL11 ∈ edgeL11 → vertL11
@axm_targetL11_def partition(targetL11, {e11_1 ↦ T11_2}, {e11_2 ↦ T11_3}, {e11_3 ↦ T11_4}, {e11_4 ↦ T11_3}, {e11_5 ↦ T11_5}, {e11_6 ↦ T11_4}, {e11_7 ↦ T11_5})
@axm_attrNL11_type attrNL11 ∈ attrL11 → vertL11
@axm_attrNL11_def partition(attrNL11, {atr11_1 ↦ T11_2}, {atr11_2 ↦ T11_3})
@axm_attrVL11_type attrVL11 ∈ attrL11 → ℙ(ℕ)
@axm_attrVL11_def partition(attrVL11, {atr11_1 ↦ ℕ}, {atr11_2 ↦ ℕ})
@axm_tL11_V tL11_V ∈ vertL11 → vertT
@axm_tL11_V_def partition(tL11_V, {T11_1 ↦ History}, {T11_2 ↦ Commit}, {T11_3 ↦ CGNode}, {T11_4 ↦ CGNode}, {T11_5 ↦ CGNode}, {T11_6 ↦ CG})
@axm_tL11_E tL11_E ∈ edgeL11 → edgeT
@axm_tL11_E_def partition(tL11_E, {e11_1 ↦ Release}, {e11_2 ↦ Loc}, {e11_3 ↦ Vis}, {e11_4 ↦ WW}, {e11_5 ↦ RF}, {e11_6 ↦ LockC}, {e11_7 ↦ Loc})
@axm_tL11_A tL11_A ∈ attrL11 → attrT
@axm_tL11_A_def partition(tL11_A, {atr11_1 ↦ ID}, {atr11_2 ↦ ID})

event CommitLoop2
any
mV
mE
mA
e11_12
e11_14
delE
where
@grd_mV mV ∈ vertL11 → vertG
@grd_mE mE ∈ edgeL11 → edgeG
@grd_mA mA ∈ attrL11 → attrG
@grd_delE delE = [{e11_6}]
@grd_newe11_12 e11_12 ∈ ℕ\edgeG
@grd_newe11_14 e11_14 ∈ ℕ\edgeG
@grd_e11_12_e11_14 e11_12 ≠ e11_14
@grd_vertices ∀v · v ∈ vertL11 ⇒ tL11_V(v) = tG_V(mV(v))
@grd_edges ∀e · e ∈ edgeL11 ⇒ tL11_E(e) = tG_E(mE(e))
@grd_attrs ∀a · a ∈ attrL11 ⇒ tL11_A(a) = tG_A(mA(a))
@grd_srctgt ∀e · e ∈ edgeL11 ⇒ mV(sourceL11(e)) = sourceG(mE(e)) ∧ mV(targetL11(e)) = targetG(mE(e))
@grd_attrComp ∀a · a ∈ attrL11 ⇒ mV(attrNL11(a))=attrNG(mA(a)) ∧ attrVG(mA(a)) ∈ ℕ
then
@act_edgeG edgeG := (edgeG \ delE) ∪ {e11_12, e11_14}
@act_sourceG sourceG := (delE ⩤ sourceG) ∪ {e11_12 ↦ mV(T11_5), e11_14 ↦ mV(T11_6)}
@act_targetG targetG := (delE ⩤ targetG) ∪ {e11_12 ↦ mV(T11_3), e11_14 ↦ mV(T11_4)}
@act_tG_E tG_E := (delE ⩤ tG_E) ∪ {e11_12 ↦ RW, e11_14 ↦ Free}
end

C.2.5 CommitLoop2_Release rule

See Fig. 65 for graph notation.

sets
vertL12
edgeL12
attrL12
constants
T12_1 T12_2 T12_3 T12_4 T12_5
e12_1 e12_2 e12_3
atr12_1 atr12_2
sourceL12
targetL12
attrNL12
attrVL12
tL12_V
tL12_E
tL12_A
axioms
@axm_vertL12 partition(vertL12, {T12_1}, {T12_2}, {T12_3}, {T12_4}, {T12_5})
@axm_edgeL12 partition(edgeL12, {e12_1}, {e12_2}, {e12_3})
@axm_attrL12 partition(attrL12, {atr12_1}, {atr12_2})
@axm_sourceL12_type sourceL12 ∈ edgeL12 → vertL12
@axm_sourceL12_def partition(sourceL12, {e12_1 ↦ T12_1}, {e12_2 ↦ T12_3}, {e12_3 ↦ T12_5})
@axm_targetL12_type targetL12 ∈ edgeL12 → vertL12
@axm_targetL12_def partition(targetL12, {e12_1 ↦ T12_3}, {e12_2 ↦ T12_2}, {e12_3 ↦ T12_5})
@axm_attrNL12_type attrNL12 ∈ attrL12 → vertL12
@axm_attrNL12_def partition(attrNL12, {atr12_1 ↦ T12_3}, {atr12_2 ↦ T12_5})
@axm_attrVL12_type attrVL12 ∈ attrL12 → ℙ(ℕ)
@axm_attrVL12_def partition(attrVL12, {atr12_1 ↦ ℕ}, {atr12_2 ↦ ℕ})
@axm_tL12_V tL12_V ∈ vertL12 → vertT
@axm_tL12_V_def partition(tL12_V, {T12_1 ↦ History}, {T12_2 ↦ Begin}, {T12_3 ↦ Commit}, {T12_4 ↦ CG}, {T12_5 ↦ CGNode})
@axm_tL12_E tL12_E ∈ edgeL12 → edgeT
@axm_tL12_E_def partition(tL12_E, {e12_1 ↦ Release}, {e12_2 ↦ Next}, {e12_3 ↦ Loc})
@axm_tL12_A tL12_A ∈ attrL12 → attrT
@axm_tL12_A_def partition(tL12_A, {atr12_1 ↦ ID}, {atr12_2 ↦ ID})

event CommitLoop2_Release
any
mV
mE
mA
e12_4
e12_6
e12_7
delE
where
@grd_mV mV ∈ vertL12 → vertG
@grd_mE mE ∈ edgeL12 → edgeG
@grd_mA mA ∈ attrL12 → attrG
@grd_delE delE = [{e12_1, e12_3}]
@grd_newe12_4 e12_4 ∈ ℕ\edgeG
@grd_newe12_6 e12_6 ∈ ℕ\edgeG
@grd_newe12_7 e12_7 ∈ ℕ\edgeG
@grd_e12_4_e12_6 e12_4 ≠ e12_6
@grd_e12_4_e12_7 e12_4 ≠ e12_7
@grd_e12_6_e12_7 e12_6 ≠ e12_7

@grd_vertices $\forall v \cdot v \in$ vertL12 $\Rightarrow$ tL12_V(v) = tG_V(mV(v))
@grd_edges $\forall e \cdot e \in$ edgeL12 $\Rightarrow$ tL12_E(e) = tG_E(mE(e))
@grd_attrs $\forall a \cdot a \in$ attrL12 $\Rightarrow$ tL12_A(a) = tG_A(mA(a))
@grd_srctgt $\forall e \cdot e \in$ edgeL12 $\Rightarrow$ mV(sourceL12(e)) = sourceG(mE(e)) $\land$ mV(targetL12(e)) = targetG(mE(e))
@grd_attrComp $\forall a \cdot a \in$ attrL12 $\Rightarrow$ mV(attrNL12(a))=attrNG(mA(a)) $\land$ attrVG(mA(a)) $\in \mathbb{N}$
then
@act_edgeG edgeG := (edgeG $\setminus$ delE) $\cup$ {e12_4, e12_6, e12_7}
@act_sourceG sourceG := (delE $\lhd$ sourceG) $\cup$ {e12_4 $\mapsto$ mV(T12_1), e12_6 $\mapsto$ mV(T12_5), e12_7 $\mapsto$ mV(T12_5)}
@act_targetG targetG := (delE $\lhd$ targetG) $\cup$ {e12_4 $\mapsto$ mV(T12_2), e12_6 $\mapsto$ mV(T12_5), e12_7 $\mapsto$ mV(T12_5)}
@act_tG_E tG_E := (delE $\lhd$ tG_E) $\cup$ {e12_4 $\mapsto$ OP, e12_6 $\mapsto$ Done, e12_7 $\mapsto$ Vis}
end