A Graph Grammar formalism for Software Transactional Memory Opacity

Diogo J. Cardoso
Federal University of Pelotas
Pelotas, RS, Brazil
diogo.jcardoso@inf.ufpel.edu.br

Luciana Foss
Federal University of Pelotas
Pelotas, RS, Brazil
lfoss@inf.ufpel.edu.br

Andre R. Du Bois
Federal University of Pelotas
Pelotas, RS, Brazil
dubois@inf.ufpel.edu.br

ABSTRACT

In order to check the correctness of Transactional Memory (TM) systems, a formal description of the implementations guarantees is necessary. There are many consistency conditions for transactional memory (TM), one of the most common is *opacity*. In this paper we present a formal framework to prove opacity on TM *histories* using a Graph Transformation System (GTS). We explore the connection between a history definition, a sequence of actions to the TM, and a GTS derivation, a sequence of direct graph transformations. Thus, creating a framework capable of observing the property of opacity on TM histories.

CCS CONCEPTS

• **Theory of computation → Verification by model checking**; • **Software and its engineering → Semantics**; *Parallel programming languages*;

KEYWORDS

Formal Semantics, Graph Grammar, Transactional Memory, Concurrent Programming, Model Checking

ACM Reference Format:
Diogo J. Cardoso, Luciana Foss, and Andre R. Du Bois. 2019. A Graph Grammar formalism for Software Transactional Memory Opacity. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, SALVADOR, Brazil.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

Transactional Memory (TM) provides a high level concurrency control abstraction. At language level, TM allows the programmer to define certain blocks of code to run atomically [20], 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 on the case of conflicts. There are benefits of using TM over lock based systems, such as, composability [18], scalability, robustness [33] and increase in productivity [27]. Now a days, there are several proposals of implementations of TM: exclusively Software [31], supported by Hardware [20], or even hybrid approaches [9, 24].

TM allows developing of a program and reasoning about its correctness as if each atomic block executes as a *transaction*, atomically and without interleaving with other blocks, and even though in reality the blocks can be executed concurrently. This is guaranteed by the formalization of observational refinement [1]: every behavior a user can observe of a program using a TM implementation can also be observed when the program uses an abstract TM that executes each block atomically. The TM runtime is responsible to ensure correct management of shared state, therefore, correctness of TM clients depend on a correct implementation of TM algorithms [23].

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 running of the same set. Several correctness criteria were proposed in the literature [11, 12, 16, 22, 25] and they rely on the concept of transactional histories. Recent works on *formal definitions* for TM focuses on consistency conditions [4, 13, 23, 32], fault-tolerance [21, 26], scalability [6, 28].

There are various techniques for formalization of a language. *Graph transformation systems* (GTSs) are a flexible formalism for the specification of complex systems, taking into account object-orientation, concurrency, mobility and distribution of systems [14]. They are well-suited for applications in which states involves many types of elements, different relations between them, and applications in which behavior is essentially data-driven.

The goal of this paper is to explore graph grammars as a proof verification for TM. The focus will be on consistency conditions, more specifically, opacity [16]. The main goal is to describe a GTS that can be used as automated validation method for TM histories, and in this paper we describe a definition to observe the property of opacity. In our approach we specify the evaluation of a TM history using executable graph production rules as provided by the GROOVE [29] tool. GROOVE also provides a model checker that allows the statement of properties on graph patterns that define the enabling of a rule.

The remainder of this paper is organized as follows. Section 2 describes the background deemed necessary for this paper, where we present a brief definition of TM, its properties, and a summary of the GTS formalism. In Section 3, we present the formal definition for our proposed extension of GTS. Section 4 describes a full example using the proposed formalism. Section 5 presents some related work. Section 6 concludes the paper.

SBLP 2019, Sept. 23–27, 2018, SALVADOR, Brazil

ACM ISBN 000-0-0000-0000-0/0/00…$15.00
https://doi.org/00.0000/0000000.0000000

2 BACKGROUND

This section describes the main background used for this paper. First, Section 2.1 contains the definition of Transactional Memory and its properties, in the context of this paper. Second, Section 2.2 shows how Graph Grammars provide means of language formalization.

2.1 Transactional Memory

Transactional Memory (TM) 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. A transaction can issue any number of operations on *transactional objects* (*t-objects*), and then it can either *commit* or *abort*. When a transaction T commits, all its operations appear as if they were executed instantaneously (atomically), at some point within the lifespan of T. When T aborts, however, all its operations are rolled back, and their effects are not visible to any other transactions [17].

A TM can itself be implemented as a shared object with operations that allow processes to control transactions. The transactions themselves, as well as t-objects, are then "hidden" inside the TM. They are, in fact, abstract notions, and it is up to the TM implementation to map them to data structures stored in local variables and base or shared objects. Conflict detection between concurrent transactions may be eager, if a conflict is detected the first time a transaction accesses a value, or lazy when it occurs only 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 occurs only when committing.

A process can access t-object only 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*.

A TM is then a shared object with the following operations (where $x_m \in TObj$ and $T_k \in Trans$):

- $inv_k(x_m, op, args)$, which *invokes* an operation op on t-object x_m using the arguments $args$ within transaction T_k, and expects a matching operation *response*;
- $ret_k(x_m, op, val)$ is the response of a invocation issued by transaction T_k, where val is the value returned by the operation.
- $tryC_k$, which attempts to commit transaction T_k, and returns either A_k or C_k;
- $tryA_k$, which aborts transaction T_k, and always returns value A_k.

The special value A_k that can be returned by the operations of a TM indicates that transaction T_k has been aborted. Value C_k returned by operation $tryC_k$ means that transaction T_k has indeed been committed. A TM can also provide an operation *init*. Every operation of a TM (except for *init*) can return a special value A_k, for any transaction T_k. Hence, the TM is allowed to force T_k to abort. This is a crucial feature, because if only $tryA$ could return A_k, then the TM could not use any optimistic concurrency control scheme, which would hamper significally the parallelism of transactions.

2.1.1 Correctness issues. To an application, all operations of a *committed* transaction appear as if they were executed instantaneously at some single and unique 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 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.

2.1.2 Properties of TM. Several safety conditions for TM were proposed in the literature, such as opacity [16], VWC [22], TMS1 and TMS2 [11] and markability [25]. All these conditions define indistinguishably criteria and set correct histories generated by the execution of TM.

Given a TM algorithm, it has the property of opacity if, all histories produced by it are legal and preserves the real time ordering of execution, i.e. if a transaction T_i commits and updates a variable x before T_j starts, then T_j cannot observe the old state of x. Guerraoui and Kapalka [16] define formally opacity and provide a graph-based characterization of such property in a way that an algorithm is opaque only if the graph built from algorithm histories structure is acyclic.

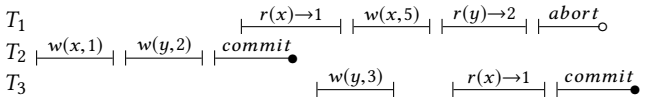

Figure 1: An opaque history H_1 [16].

To illustrate the definition of opacity, consider the following history H_1, of three transactions accessing two registers (x and y), corresponding to the execution in Figure 1.

$$\begin{aligned} H_1 \quad = \quad & \langle write_2(x,1), write_2(y,2), tryC_2, \\ & inv_1(x, read, \bot), C_2 \\ & inv_3(y, write, 3), ret_1(x, read, 1) \\ & inv_1(x, write, 5), ret_3(y, write, ok) \\ & ret_1(x, write, ok), inv_1(y, read, \bot) \\ & inv_3(x, read, \bot), ret_1(y, read, 2), tryC_1 \\ & ret_3(x, read, 1), tryC_3, A_1, C_1 \rangle. \end{aligned}$$

Definition 1. *A history H is opaque if there exists a sequential history S equivalent to some history in set Complete(H) (i.e., does not contain any live transaction), such that (1) S preserves the real-time order of H, and (2) every transaction $T_i \in S$ is legal in S (i.e., respects the sequential specifications of all the shared objects).*

2.2 Graph Grammar

Graphs and graph transformations represent the core of most visual languages [3]. 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. Then, the events occurring in the system, which are responsible for the evolution from one state into another, are modelled as the application of 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 [2].

Graph transformation systems (GTSs) are a flexible formalism for the specification of complex systems, that may take into account aspects such as object-orientation, concurrency, mobility and distribution [14]. GTSs are specially well-suited for the formal specification of applications in which states involves not only many types of elements, but also different types of relations between them. Also, applications in which behavior is essentially data-driven, that is, events are triggered by particular configurations of the state [5].

A GTS consists of a set of rewriting rules, also called graph productions [30]. The states of a GTSs are modeled by graphs and state changes, or events, which can be transformed by rewriting rules. Many reactive systems are examples of this class of applications, like protocols for distributed and mobile systems, biological systems, and many others. On top of the complex states and reactive behavior, concurrency and non-determinism are essential in this area of applications, in the sense that many events may happen concurrently, if they are all enabled, and the choice of occurrence between conflicting events is non-deterministic [5].

2.2.1 GTS. Following the style of [15, 19], a *graph* is a tuple $\langle N, E, L\rangle$ consists of a finite set N of nodes, a finite set $E \subseteq N \times N$ of edges, and a labelling function L of nodes and edges. $\mathcal{G}$ is the set of all graphs, ranged over G, H, etc. A graph matching $m : \mathcal{G} \to \mathcal{G}$ is a graph morphism that preserves node and edge labels. A graph transformation rule $p \in R$ specifies how the system evolves when going from one state to another: it is identified by its name ($Np \in \mathcal{N}$, where $\mathcal{N}$ is a global set of rule names) and consists of a left-hand side graph (L_p), a right-hand side graph (R_p), and a set of so-called negative application conditions (NAC_p, which are super-graphs of L_p).

Definition 2 (Graph Production System). *A graph production system (GPS) is a tuple $P = \langle R, I\rangle$ that consists of a set of graph transformation rules R and graph I, the initial state.*

The application of a graph transformation rule p transforms a graph G, the source graph, into a graph H, the target graph, by looking for an occurrence of L_p in G (specified by a graph matching m) and then by replacing that occurrence with R_p, resulting in H. Such a rule application is denoted as $G \to_{p,m} H$. Each GPS $P = \langle R, I\rangle$ specifies a (possibly infinite) state space which can be generated by repeatedly applying the graph transformation rules on states, starting from the initial state I.

Definition 3. *A GTS $T = \langle S, \to, I\rangle$ generated by $P = \langle R, I\rangle$ consists of a set $S \subseteq \mathcal{G}$ of graphs representing states, an initial state $I \in S$, and a transition relation $\to \in S \times R \times [\mathcal{G} \to \mathcal{G}] \times S$, such that $\langle G, p, m, H\rangle \in \to$ iff there is a rule application $G \xrightarrow{p,m} H'$ isomorphic to H.*

3 DEFINING TM USING GTSS

This chapter presents the definition of transactional memory actions using GTSs and a view of the opacity property. First, we need a syntax for the TM, and some definitions that will used as criteria for the property of opacity.

A *program* $P = T_1 \| \ldots \| T_n$ is a parallel composition of *transactions*, where each transaction T_n can execute one *command* C_n at a time, representing the current state of execution of T_n. Every command C_n has access to a set of *local variables* $Lvar$, which are exclusive to the specific transaction; for simplicity, these are integer-valued. Transactions have access to a *transactional memory* (TM), which manages a fixed collection of *t-variables* $Tvar$. The syntax of the commands C of a given transaction n is as follows ($e \in \mathbb{Z}$):

C = $init_n$ | $Lvar_n$:= $Tvar$.read() | $Tvar$.write$_n(e)$ | $tryC_n$ | $commit_n$ | $abort_n$

Definition 4 (Version Order). *The version order of a $x \in$ Reg in a TM history H is any total order $\ll_x$ on the set of transactions H that: (1) are committed or commit-pending, and (2) write to x.*

Definition 5 (Real-Time Order). *For every TM history H, there is a partial order $\prec_H$ that represents the real-time order of transactions in H. Given any 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, then $T_k \prec_H T_m$.*

According to Guerraoui and Kapałka [17], to avoid dealing with the initial values of t-variables separately from the values written to those t-variables by transactions, a "virtual" committed initializing transaction T_0 needs to be introduced. T_0 writes value 0 to every t-variable (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 following the rules:

(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 reads from 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 $T_i \to T_k$) 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 "from real-time") and denoted $T_i \xrightarrow{rt} T_k$;
   (b) If T_k reads from T_i; then the edge is labelled rf and denoted $T_i \xrightarrow{rf} T_k$;
   (c) If, for some variable $x \in$ Reg, $T_i \ll_x T_k$; 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: (a) $T_m \ll_x T_k$, and (b) T_i reads x from T_m; then the edge is labelled rw (from "read before write") and denoted $T_i \xrightarrow{rw} T_k$;

Theorem 3.1 (Graph characterization of opacity). *A consistent TM history H that has unique writes is opaque (Def. 1) if, and*

only if, there exists a version order function $V_{\ll}$ in H such that graph $OPG(H, V_{\ll})$ is acyclic.

Proof. $\Rightarrow$ Let H be any opaque TM history that has unique writes. By final-state opacity, there exists a sequential history S equivalent to some completion of H, such that S preserves the real-time order of H and every transaction T_i in S is legal in S. For every variable $x \in \mathsf{Reg}$, let $\ll_x$ denote the following version order of x in H: for all transactions T_i and T_k in H that are committed or commit-pending and that write to x, if $T_i \prec_S T_k$ or $i = 0$, then $T_i \ll_x T_k$. Let $V_{\ll}$ be the version order function in H such that, for every variable $x \in \mathsf{Reg}$, $V_{\ll}(x) = \ll_x$. It is possible to show that graph $G = OPG(H, V_{\ll})$ is acyclic. □

3.1 GTS Definition

Using the rules of construction of an opaque graph $OPG(H, V_{\ll})$, it is possible to define a GTS that will generate the graph associated with a given history H. Therefore, the result graph can show if the history H is opaque, by its acyclicity. We will be using the application GROOVE [29] for the modeling of our definition[1]. GROOVE is based on a graph representation of system states and on a representation of state updates via graph transformations. Graph production rules specify both matching patterns and negative application conditions (NAC). Graph matching is used to select the pattern in the host graph that has to be rewritten into a new graph (obtained by deleting/adding/merging nodes and edges). To specify data fields ranging over basic types like booleans, integers, and strings, we can use node attributes. Attributes are treated as special edges that do not point to a standard node, but to a node that corresponds to a data value.

Now we need to define how this sequence of actions is evaluated. For our definition we use a node to represent each action separately, and any information necessary to the action is carried with it. The sequential view of the history is represented with a operator *GO*, that follows an ordered path at each step of evaluation. The Figure 2 shows an example of a state of the operator *GO*, which is connected to the next action which will be evaluated.

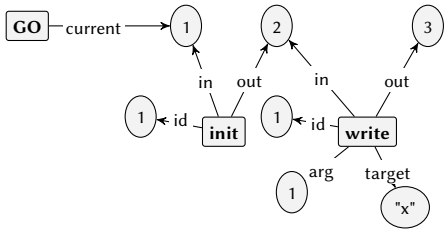

Figure 2: Graph representation of the *GO* operator.

Each action in history H is connected to two fixed numbers, which will always be in an ascending order. The edge *in* represents the current action to be evaluated: the matching of the *GO* operator and the unique action connected to the *in* edge, results in a step of evaluation. At the end of every step, the *GO* operator moves to the next number in the sequence, by using the *out* edge of the current action. The complete type definition of the grammar can be seen in Figure 3.

[1]The entire grammar definition can be found here: https://github.com/diogocrds/tmgg

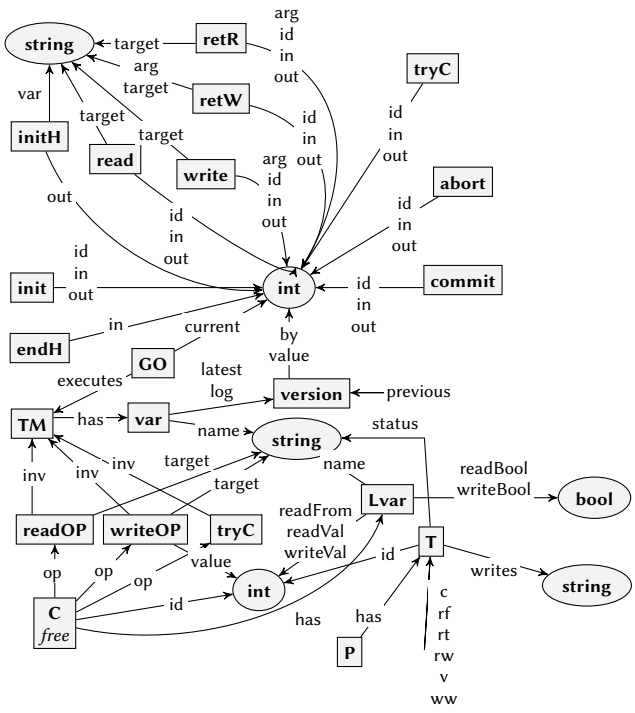

Figure 3: Type definition.

3.1.1 History first ant last operation. The first step of evaluation of any history H is given by the action *initH*, the production rule can be seen in Figure 4.

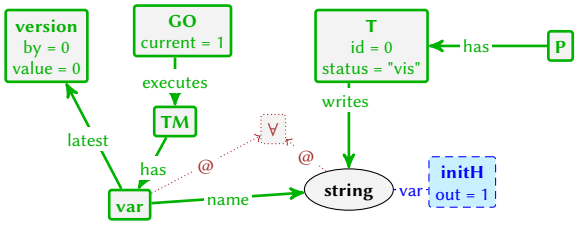

Figure 4: Production rule of *initH*.

This action marks the beginning of a history, it serves the purpose of creating the objects necessary for the evaluation of the following operations, these objects are: *TM* and its variables, the *GO* operator, and graph P with a transaction T_0. The action node *initH* carries a list of variables that will be used in this specific history, the list is used as reference for initiating all variables in the *TM*, as if they were written by T_0. This step consumes the node *initH*.

Now for the final step of evaluation of history H, an action *endH* is used, as seen in Figure 5. This action consumes every control node used in evaluation: *TM* and its variables, the operator *GO*, every command *C* left, and the node *P* of graph (meaning that the

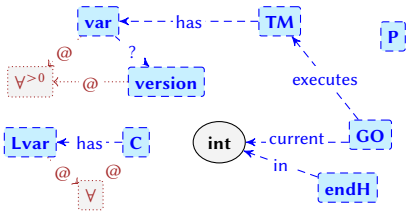

Figure 5: Production rule of *endH*.

program has ended). The node *endH* is consumed as well. Therefore, the only objects left at the end of evaluation of a history H are the transaction nodes T, and if this resulting graph is acyclic, then history H has the property of opacity.

3.1.2 Init Operation. The first operation of any transaction k in H should be an $init_k$ command. The production rule $init_k$ can be seen in Figure 6.

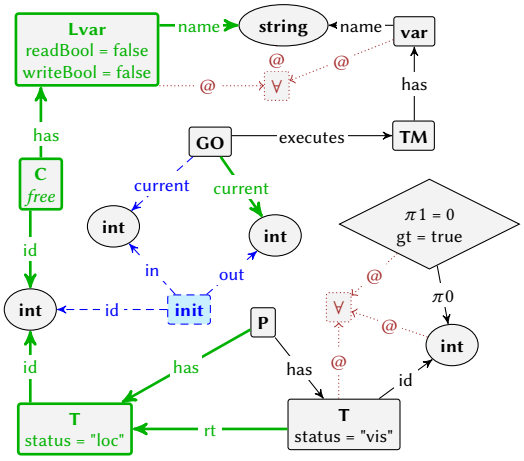

Figure 6: Production rule of $init_k$ action.

Every transaction i that has committed or is commit-pending, is connected to k ($i \neq k$) by an edge that represents the "real-time" order (tag rt), meaning that the changes made in i are visible to k. The operation $init_k$ creates a new node C_k for future commands on the same transaction, and a node T with an attribute *id* equals to k is added to the OPG graph. The history action node *init* is consumed in the step of evaluation.

3.1.3 Write Command. A write command can be denoted as $write_k(x, v)$ or $inv_k(x, write, v)$, the first is a complete representation of the action (its invocation and the successful return), and the second represents only the invocation, meaning the successful return ($ret_k(x, write, ok)$) has not happened yet. For the GTS definition, the invocation and its return are interpreted separately, therefore there are two production rules for a write command (Figures 7 and 8).

An invocation $inv_k(x, write, v)$ can only be evaluated if a flag $free$ is connected to C_k, if it is, the flag is consumed in the action. A write command connects C_k to the TM with a *writeOP* node that has the value and target of the intended write command. The

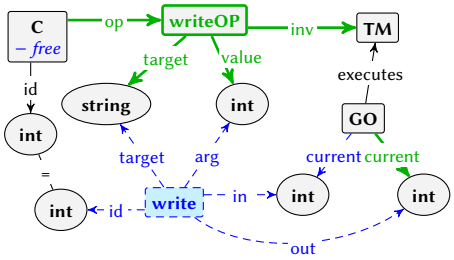

Figure 7: Graph representation of $inv_k(x, write, v)$ action.

node representing the current history action *write* is consumed. The *GO* operator moves to the next action on its list. This action has no effect on the OPG graph.

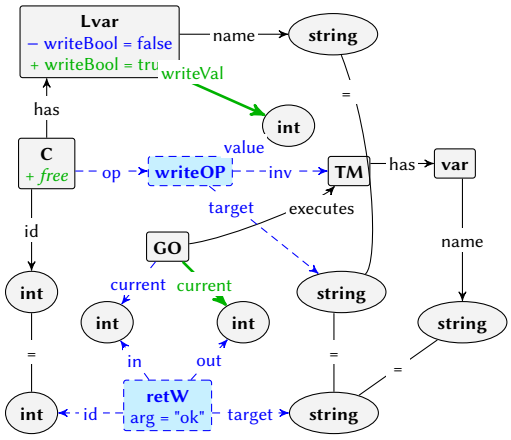

Figure 8: Graph representation of $ret_k(x, write, ok)$ action.

The return action ($ret_k(x, write, ok)$) of a write command can only be performed if C_k is connected to the TM by a *writeOP* node, i.e., C_k has performed a write invocation. The connecting node *writeOP* between C_k and TM is consumed. A successful write return means that C_k can perform other actions, so the flag $free$ is created on C_k. A boolean *true* is assigned to the flag *writeBool* of the local variable x in C_k, this is used to know if a transaction has written to a specific variable. The *GO* operator moves to the next action on its list, and the current history action $retW$ is consumed. This action also has no effect on the OPG graph.

3.1.4 Read Command. Similarly to the write command, a read command can be denoted as $read_k(x)$ as a complete operation, or $inv_k(x, read, \perp)$ and $ret_k(x, read, v)$, but for our definition we use only the second notation. The production rules for both the invocation and return of a read command can be seen in Figures 9 and 10.

An invocation $inv_k(x, read, \perp)$ can only be evaluated if C_k has the flag $free$, which is consumed in the action. The value of the variable in question (*target*) by definition has been assigned by another transaction Ti (possibly T_0, as an initial value), so this action is seen as T_k *reads from* T_i. The result of a read invocation action connects C_k to the TM. The *GO* operator moves to the next

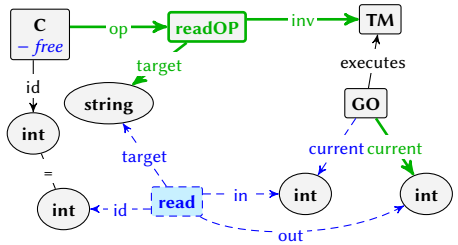

Figure 9: Graph representation of $inv_k(x, read, \perp)$ action.

action on its list, and the current history action *read* is consumed. This action has no effect on the OPG graph.

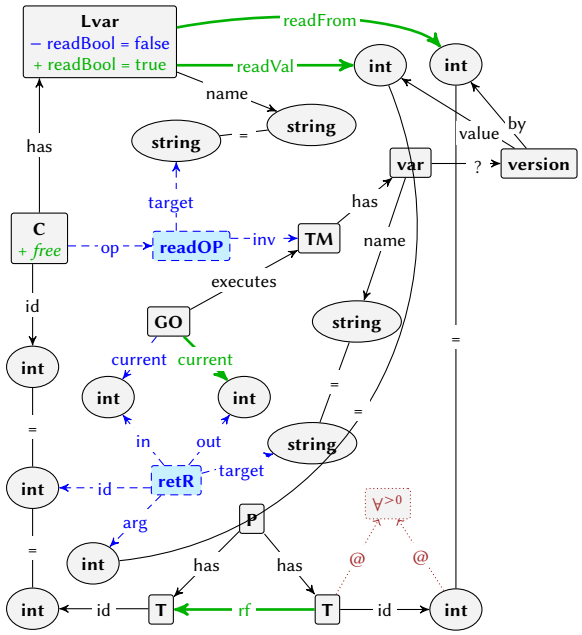

Figure 10: Graph representation of $ret_k(x, read, v)$ action.

The return action of a read operation (Figure 10), can only be performed if C_k is connected to the TM by a *readOP* node, i.e., C_k has performed a read invocation. The node *readOP* is consumed in this action. The received value of the variable (*target*) is assigned to the local variable in C_k, and the boolean flag *readBool* has now the value of *true*. The *GO* operator moves to the next action on its list, and the current history action *retR* is consumed.

The nodes T_i and T_k, in the OPG graph, are part of the set of transaction nodes, all transaction exist under program P. The action of reading a variable from T_i, means that the nodes T_i and T_k are now connected by an *rf* edge.

3.1.5 Commit Operation. A commit operation is divided into two phases: a *tryCommit*, which means that the transaction will not perform any other operations to the TM; and its result, to *commit* or *abort* the transaction. The production rules for the *tryCommit* and *commit* actions are shown in Figures 11, 12. Here, we omit the *abort* action to save space, its functionality is simply to roll back the values assigned to the TM, because any changes made by the transactions are not visible anymore.

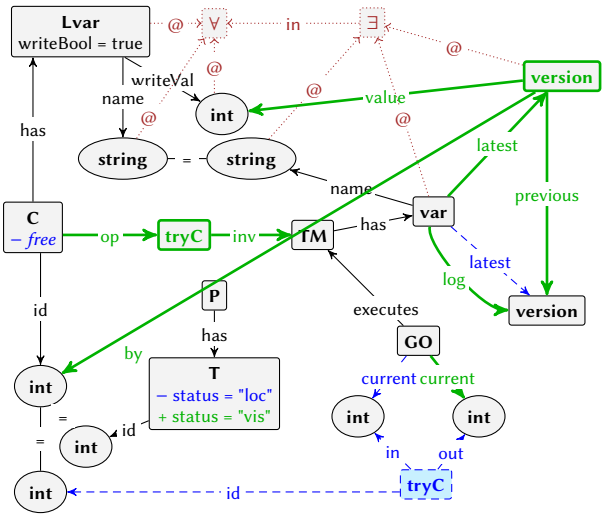

Figure 11: Graph representation of $tryC_k$ action.

An operation $tryC_k$ makes the transaction k visible to other transactions, i.e., the TM now has a reference to transaction T_k in any variable that was written it. The value that transaction C_i, who wrote into x previously, is stored in C_k to allow an undo action in case of *abort*. The history action node *tryC* is consumed in the step of evaluation. In the OPG graph, the transaction T_k has now the tag *vis*.

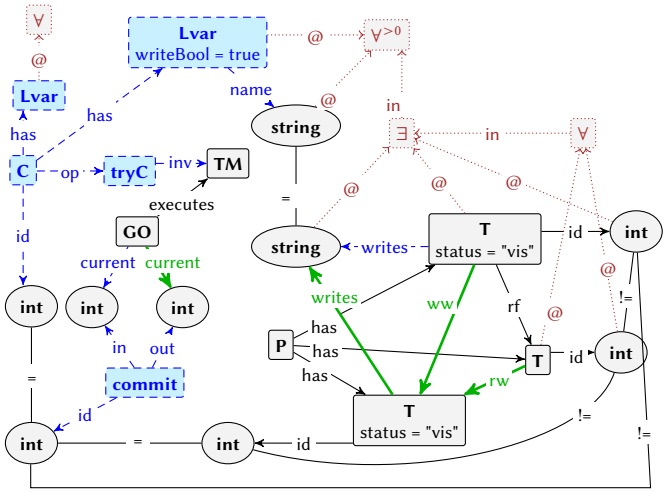

Figure 12: Graph representation of C_k action.

The operation *commit* marks the final action of a successful transaction, the changes to the TM are maintained. The node C_k is consumed in the evaluation, as well as any current connection to the TM. In the OPG graph, if C_k has made a write to any variable

x, the node of the previous transaction T_i who wrote into the same variable (possibly T_0) now has an edge with the tag of *ww* between T_i and T_k ("write-before-write").

3.2 Observing Opacity

Based on our grammar definition, we can generate a graph that represents the dependencies between transactions in a given history. Now, we need to test if this graph is acyclic, so that we can conclude whether or not the property of opacity can be observed in the history used as input. For this, we use the GROOVE model checker for a CTL temporal logic on transitions. The model checker operates on the labelled transitions system (LTS) induced by a set of graph productions. CTL propositions are defined on top of transition names. This way it is possible to state properties on graph patterns that define the enabling of a rule [10].

Figure 14 shows the production rule that is executed after the evaluation of the history has ended (the node P must not be present). A match occurs for this rule every time that two different transactions are connected by a dependency relation (edge '?' means any type of edge). The first transaction must have the flag 'c' (*current*) that is consumed, and receives a new flag 'v' (*visited*) that must not exist already. The result is simply the same graph with flags of *visited* and *current*.

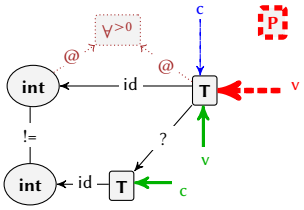

Figure 13: Production rule for loop marking.

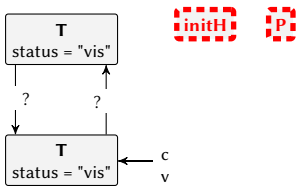

Figure 14: Production rule *cyclic*.

After marking the entire graph, the LTS generated is a tree of states of the different paths taken. If the production rule *cyclic* can be matched to any state of this tree, then the history is not opaque. The test is made by a CTL formula

```
AG !cyclic
```

which means that, for every state and every configuration of the graph, the production rule `cyclic` (in this case, considered a graph condition) must not be matched.

Finally, if we regard an action in the history as a production rule (transition), a sequence of actions can be seen as a derivation. By operating on the LTS, it is possible to check whether a sequence of transitions result in a acyclic graph or not. Therefore, we can observe the property of opacity on any given history as input to the GTS.

4 EXAMPLE

Now to demonstrate the proposed GTS in action, we run a few examples of histories through GROOVE's model checker. Figure 15 presents three different histories, their full graph definition are included in the *github* repository referred previously.

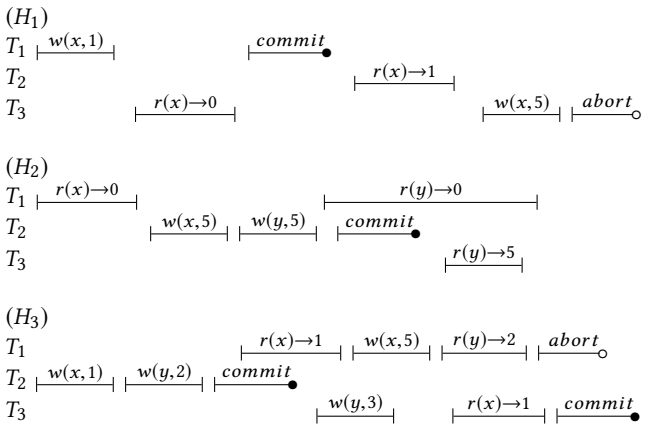

Figure 15: Three opaque histories H_1, H_2 and H_3.

The generated OPG graphs can be seen in Figures 16 and 17.

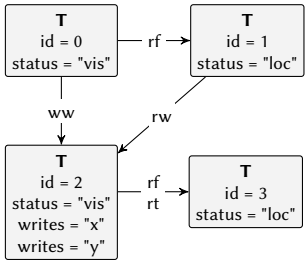

Figure 16: OPG graph resulted from H_2.

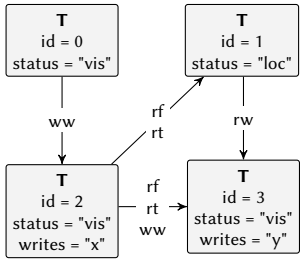

Figure 17: OPG graph resulted from H_3.

At this point of the evaluation the production rules for loop marking are applied, which simply results in the same graph with a few extra flags on the nodes. Finally, the model checker can be used to evaluate the states of the simulation to check if `cyclic` is applicable, and as shown in Figures 16 and 17, it is not, so we can conclude that the input history in question is indeed opaque.

5 RELATED WORKS

There have been previous work that focus on formal definitions of properties of TMs, some can be related to our proposal for being properties that are related to the history of a TM program. The work of Khyzha et al. [23] proposes a notion of transactional *data-race freedom* (DRF) that takes into account selective transactional fence placements. The Fundamental Property is formalized using observational refinement: if a program is DRF under strong atomicity (formalized as transactional sequential consistency [7, 8]), then all its executions are observationally equivalent to strongly atomic ones. Furthermore, their work prove that the Fundamental Property holds under a certain condition on the TM, generalizing opacity [16, 17], which is called strong opacity. So, similarly to non-transactional memory models, the programmer writing code that has no data races according to this notion never needs to reason about weakly atomic semantics.

Bushkov et al. [4] introduce a new weak consistency condition, called *weak adaptive consistency*, and prove the so called PCL theorem: in transaction systems it is possible to ensure and strict disjoint-access-parallelism (Parallelism), weak adaptive consistency (Consistency) and obstruction freedom (Liveness). To circumvent the impossibility result it is sufficient to weaken just one of the three requirements.

6 CONCLUSIONS

In this paper we have proposed an application of Graph Transformation Systems to the validation of Transactional Memory histories, using the tool GROOVE for definition and model checking. The sequence of actions made in a TM history can be interpreted as transitions in the state of a graph. Therefore, by using these transitions to generate a graph that represents the dependencies between transactions, it is possible to verify if the input history has the property of opacity.

This paper serves as an initial proposition on the use of Graph Grammars to prove properties of TM, which we consider its main contribution. For future work we intend to explore more complex definitions of properties (strong opacity, atomicity, etc), and also to bring this idea to the algorithm level, e.g., where we can prove that a certain TM algorithm only generates opaque histories. There are also some GG features that would be interesting to explore in the context of TM, like transactions as part of the graph definition (unstable nodes that represent atomic actions), or even translations to event-b (theorem proving method for GG).

REFERENCES

[1] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2013. A programming language perspective on transactional memory consistency. In *2013 ACM symposium on Principles of distributed computing*. 309–318.

[2] Paolo Baldan, Andrea Corradini, Fernando Luís Dotti, Luciana Foss, Fabio Gadducci, and Leila Ribeiro. 2008. Towards a notion of transaction in graph rewriting. *Electronic Notes in Theoretical Computer Science* 211 (2008), 39–50.

[3] M. Bardohl, R.and Minas, G. Taentzer, and A. Schurr. 1999. APPLICATION OF GRAPH TRANSFORMATION TO VISUAL LANGUAGES. *Handbook of graph grammars and computing by graph transformation* 2 (1999), 105.

[4] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. 2018. The PCL Theorem: Transactions cannot be Parallel, Consistent, and Live. *Journal of the ACM (JACM)* 66, 1 (2018), 2.

[5] Simone André da Costa Cavalheiro, Luciana Foss, and Leila Ribeiro. 2017. Theorem proving graph grammars with attributes and negative application conditions. *Theoretical computer science* 686 (2017), 25–77.

[6] Austin T Clements, M Frans Kaashoek, Eddie Kohler, Robert T Morris, and Nickolai Zeldovich. 2017. The scalable commutativity rule: designing scalable software for multicore processors. *Commun. ACM* 60, 8 (2017), 83–90.

[7] Luke Dalessandro and Michael L Scott. 2009. Strong isolation is a weak idea. In *TRANSACTâĂŹ09: 4th Workshop on Transactional Computing*.

[8] Luke Dalessandro, Michael L Scott, and Michael F Spear. 2010. Transactions as the foundation of a memory consistency model. In *International Symposium on Distributed Computing*. Springer, 20–34.

[9] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *ACM Sigplan Notices*, Vol. 41. ACM, 336–346.

[10] Giorgio Delzanno and Riccardo Traverso. 2013. Specification and validation of link reversal routing via graph transformations. In *International SPIN Workshop on Model Checking of Software*. Springer, 160–177.

[11] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2009. Towards formally specifying and verifying transactional memory. *Electronic Notes in Theoretical Computer Science* 259 (2009), 245–261.

[12] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing* 25, 5 (2013), 769–799.

[13] Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. 2015. Consistency for transactional memory computing. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Springer, 3–31.

[14] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-J rg Kreowski. 1999. *Handbook of graph grammars and computing by graph transformation*. Vol. 3. world Scientific.

[15] Rozenberg Grzegorz. 1999. *Handbook Of Graph Grammars And Comp. By Graph Transformations, Vol 2: Applications, Languages And Tools*. world Scientific.

[16] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 175–184.

[17] Rachid Guerraoui and Michał Kapałka. 2010. Principles of transactional memory. *Synthesis Lectures on Distributed Computing* 1, 1 (2010), 1–193.

[18] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[19] Reiko Heckel. 2006. Graph transformation in a nutshell. *Electronic notes in theoretical computer science* 148, 1 (2006), 187–198.

[20] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. ACM.

[21] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. 2017. Hipertm: High performance, fault-tolerant transactional memory. *Theoretical Computer Science* 688 (2017), 86–102.

[22] Damien Imbs and Michel Raynal. 2012. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science* 444 (2012), 113–127.

[23] Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. 2018. Safe privatization in transactional memory. *ACM SIGPLAN* 53 (2018), 233–245.

[24] Sanjeev Kumar, Michael Chu, Christopher J Hughes, Partha Kundu, and Anthony Nguyen. 2006. Hybrid transactional memory. In *11th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 209–220.

[25] Mohsen Lesani and Jens Palsberg. 2014. Decomposing opacity. In *International Symposium on Distributed Computing*. Springer, 391–405.

[26] Ognjen Marić. 2017. *Formal Verification of Fault-Tolerant Systems*. Ph.D. Dissertation. ETH Zurich.

[27] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A study of transactional memory vs. locks in practice. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 43–52.

[28] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. 2015. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. ACM, 217–226.

[29] Arend Rensink, Maarten De Mol, and Eduardo Zambon. 2019. GROOVE GRaphs for Object-Oriented VErification (Version 5.7.4). https://groove.cs.utwente.nl/

[30] Grzegorz Rozenberg. 1997. *Handbook of Graph Grammars and Comp*. Vol. 1. World scientific.

[31] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[32] Konrad Siek and Paweł T Wojciechowski. 2014. Zen and the art of concurrency control: an exploration of TM safety property space with early release in mind. *Proc. WTTM* 14 (2014).

[33] Jons-Tobias Wamhoff, Torvald Riegel, Christof Fetzer, and Pascal Felber. 2010. RobuSTM: A robust software transactional memory. In *Symposium on Self-Stabilizing Systems*. Springer, 388–404.