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

**Diogo J. Cardoso**[1], **Luciana Foss**[1], **Andre R. Du Bois**[1]

[1]Programa de Pós-Graduação em Computação - CDTec
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brasil

{diogo.jcardoso,lfoss,dubois}@inf.ufpel.edu.br

***Abstract.*** *With the constant research and development of Transactional Memory (TM) systems, various algorithms have been proposed, and their correctness is always an important aspect to take into account. When analyzing TM algorithms, one of the most commonly used correctness criterion is opacity, which infers that executions only observe consistent states of the shared memory. This paper proposes a formal definition to demonstrate that a given TM algorithm only generates opaque histories using a Graph Transformation System. The methodology introduced consists of translating an algorithm into production rules that manipulate the state of a graph. The proposed approach has demonstrated capability to deal with some of the complexity of TM algorithms and a case study has shown the working proof of opacity of the algorithm in question.*

***Resumo.*** *Com a constante pesquisa e desenvolvimento de sistemas de Memória Transacional (TM), vários algoritmos têm sido propostos, e sua corretude é sempre um aspecto importante a se levar em consideração. Ao analisar algoritmos de TM, um dos critérios de corretude mais comumente usados é opacidade, que infere que execuções só observam estados consistentes da memória compartilhada. Este artigo propõe uma definição formal para demonstrar que um determinado algoritmo TM só gera histórias opacas usando um Sistema de Transformação de Grafos. Uma metodologia é introduzida para traduzir um algoritmo para regras de produção que manipulam o estado de um grafo. A abordagem proposta demonstrou a capacidade de lidar com algumas das complexidades de algoritmos TM e um caso de estudo mostra o funcionamento da prova de opacidade do algoritmo em questão.*

## 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 and Dechev 2017].

*Work in progress.

†This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Transactional Memory (TM) provides a high level concurrency control abstraction. At language level, TM allows programmers to define certain blocks of code to run atomically [Herlihy and 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 and Adl-Tabatabai 2011]. There are several proposals of implementations of TM: exclusively Software [Shavit and Touitou 1997], supported by Hardware [Herlihy and Moss 1993], or even hybrid approaches [Matveev and 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 and Kapalka 2008, Lesani and Palsberg 2014] and they rely on the concept of transactional histories.

Recent works on formal definitions for TM focuses on consistency conditions [Khyzha et al. 2018, Bushkov et al. 2018], fault-tolerance [Marić 2017], and scalability [Peluso et al. 2015]. Of the several correctness criteria proposed for TM, opacity is very well defined and known. Opacity and its sub-classes use a graph characterization composed of a conflict graph that represents how the transactions relate to each other by some defined notion of conflict. A history, sequence of transactional events that have access to a shared memory, is considered correct if the conflict graph of its transactions presents no cycles.

This work aims to propose a methodology to define a Graph Transformation System (GTS) that represents a TM algorithm and demonstrate that, considering the notion of conflict introduced by [Guerraoui and Kapalka 2008], the algorithm only generates "correct" histories. As an initial result and proof of concept, a GTS was constructed and demonstrated that from a single history execution it is possible to generate its conflict graph and evaluate the correctness of the history [Cardoso et al. 2019]. The main goal of this work is to expand that idea for an entire TM algorithm that generated such history. Meaning that for a set of transactions, the aim of the methodology is to show that every execution observes a correct state of the shared memory. The main contribution of this work is a methodology to formalize complex TM algorithms and its safety property check. This approach is capable of dealing with different characteristics of TM algorithms, in terms of versioning and conflict detection, which can make it a powerful tool for their correctness verification and also a formalization that can possibly be extended to new or existing graph characterization of other safety properties. A case study of a complex algorithm that deals with partial rollbacks was explored in [Cardoso et al. 2021].

## 2. Background

This chapter briefly describes some of the background knowledge necessary for the rest of this work. For space reasons not all background explanations along with the case study are present in this text, however, they can be found in a public repository[1].

### 2.1. Transactional Memory

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

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

To realize operations in shared 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.

### 2.2. 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 and 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.

[1]Full text and case study explanations can be found in `https://github.com/diogocrds/tmgts`.

**Definition 1** (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 $T_i \xrightarrow{rw} T_k$;*

**Theorem 1** (Graph characterization of Opacity [Guerraoui and 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 and Kapałka 2010]. □

## 3. Translation to GTS

The first step of the proposed methodology is 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. First, a representation of sequential operations is needed that will compose transactions and histories used in the entire approach. If `x` is any variable, # is any number/value, transactions can have the following operations: `begin`, `write(x,#)`, `read(x)` and `tryCommit`. Conflict comes form two or more transactions executing operations on the same variable.

Figure 1 demonstrates how a transaction is represented in a graph manner. This is the **initial state** as an input to the GTS. Each operation (`begin`, `read`, `write` and `tryCommit`) is represented by a node with relevant information to the operation itself, the main node `T` represents the identifier for the transaction with an unique id. Sequential operations are connected by an directed edge `next` that represents the order in which these operations must execute. The edge `op` represents the current operation to be executed, in a approach like this every transactional operation is considered to be atomic: the response of the operation happens immediately after the invocation. In a previous work, a GTS formalism for histories where the invocations and responses are processed separately [Cardoso et al. 2019] was explored. However, because some of the correctness criteria for the TM algorithms use atomic operations, it was also chosen to be used in this approach, which in turns decreases the number of nodes in a transaction or history, making it more readable.

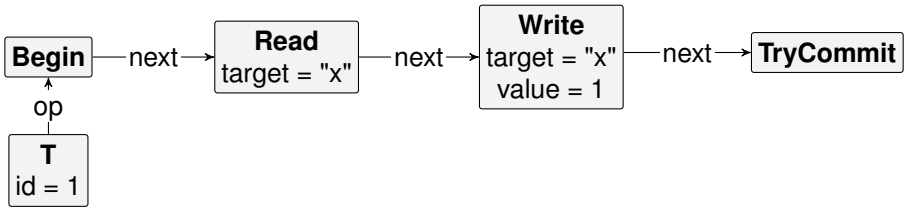


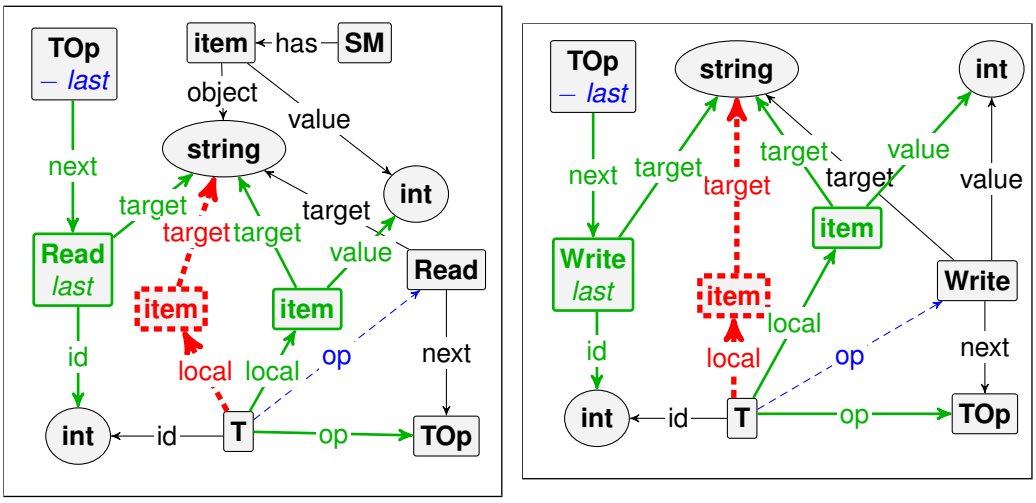

**Figure 1. Graph representation of a transaction as an input for the GTS.**

The initial state of the GTS includes the transactions (as seen in Figure 1) 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. Having these global objects from the start allows the algorithm's logic to access them at any moment. Restriction on this access is enforced by the transformation rules. Another important aspect of the GTS formalism is the type graph. This special graph will determine what nodes and edges can exist in the system, this results in a controlled behavior by the production rules.

### 3.1. TM Procedures

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

The first operations we describe are **read operation** and **write operation**. TM procedures can demonstrate different approaches in terms of versioning (eager or lazy) and conflict detection (also eager or lazy).




**(a) Lazy versioning Read**

**(b) Lazy versioning Write**

**Figure 2. Example of production rules for lazy versioning Read and Write Operations.**

In Figure 2 it is showcased a lazy versioning read and write operation. These two examples manipulate the values of the shared memory (reading or writing a specific variable) and create a new object in the last position of the history.

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

Note that so far only versioning was covered in the production rules, but conflict detection is also an important characteristic to take into consideration when designing a TM algorithm so it will be reflected in the GTS. In an algorithm with eager conflict detection, at any point if a conflict happens some transaction is likely to be aborted. This can approached by always checking the version of a variable read by the transaction. 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 observed a stable state of the system. This verification can be read as: for all local nodes `conflict` that store a value read (valRead-edge), their respective objects in the shared memory (`SM` node) must not have a different value. While the verification of conflict in a commit, read or write operation ensures that all values read are stable, the abort operation that uses the quantifier *exists* ($\exists$), can be triggered with at least one conflict. Both instances of verification are mutually exclusive, a transaction cannot commit (or read or write for that matter) and abort at the same time.

## 4. 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 Labelled Transition System (LTS) Simulation tool that GROOVE offers. Given the initial state of three transactions similar as seen in Figure 1 (in addition of the global nodes `History` and `SM`) the simulation of a *lazy-versioning* and *eager-conflict* algorithm will generate a LTS with 231 states where 70 of those are called "final". In GROOVE the LTS is visualized with a tree-like graph.

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 is labelled *start* and at the bottom the final states as green nodes labelled *result*. A final state just means that no more production rules can be applied to that state, this means that there are no more transactional operations left to be executed and the history generated by that sequence of operations is complete. 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* (a production rule that does not create or delete elements).

## 5. Correctness Criterion

The third, and final, step of the proposed methodology is to observe the correctness of the algorithm being evaluated. The correctness criteria used is mainly based on the graph

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

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

Lastly, now that the histories were generated, and from each one a conflict graph was extracted, the LTS is complete allowing the use of the Computation Tree Logic (CTL) tool in GROOVE to check for acyclicity of these conflict graphs. With some auxiliary production rules to path through the conflict graphs a condition is looked for where: following the direction of the edges results in a path that goes back to a node that was already visited. The CTL check consists of a pattern match test of the special production rules named *graph condition*. The verification is made by the formula: `AG !cyclic`, which means that, for every path following the current state, the entire path holds the property of not pattern matching the graph condition `cyclic`. If the formula holds for all states in the LTS, the condition of acyclicity is true and the algorithm only generated opaque histories.

## 6. Conclusions

In this work we presented an approach to verify opacity as a correctness criterion for transactional memory algorithms via a translation to a Graph Transformation System (GTS). We used a graph characterization of opacity [Guerraoui and Kapałka 2010] well known in the literature to demonstrate opacity to every history the algorithm generates for a given entry program. For this approach it was necessary to represent sequential operations which compose the initial state (set of transactions) and the histories. We used production rules that "execute" a full transactional operation in a single step. The Labelled Transition System (LTS) generated by the GROOVE tool represents all possible sequences of rule applications. The leafs of this tree contain a full history, thus the set of all possible rule applications represents the set of all possible histories the current algorithm generates from a set of transactions given in the initial state.

We extended a previous work, that evaluated a single history to observe its opacity using a GTS, to now prove the opacity to all the histories generated by the translated algorithm. For future work, we want to explore a more concrete initial state as input for the correctness test against an algorithm. This introduces the complexity of dealing with loops and more complex data structures other than single-value t-objects. We also have in mind the use of other TM algorithms with different approaches to detecting and dealing with conflict/versioning. Another important point that we want to address is the choice of correctness criteria, opacity was an easy answer because it already has a graph

characterization, however we want to explore other safety properties and possibly liveness properties too.

## References


Bushkov, V., Dziuma, D., Fatourou, P., and Guerraoui, R. (2018). The pcl theorem: Transactions cannot be parallel, consistent, and live. *Journal of the ACM (JACM)*, 66.

Cardoso, D., Foss, L., and Du Bois, A. (2021). A graph transformation system formalism for correctness of transactional memory algorithms. In *25th Brazilian Symposium on Programming Languages*, pages 49–57.

Cardoso, D. J., Foss, L., and Bois, A. R. D. (2019). A graph transformation system formalism for software transactional memory opacity. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages*, pages 3–10.

Guerraoui, R. and Kapalka, M. (2008). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184.

Guerraoui, R. and Kapałka, M. (2010). Principles of transactional memory. *Synthesis Lectures on Distributed Computing*, 1(1):1–193.

Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA.

Herlihy, M. and Moss, J. E. B. (1993). *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM.

Khyzha, A., Attiya, H., Gotsman, A., and Rinetzky, N. (2018). Safe privatization in transactional memory. *ACM SIGPLAN*, 53:233–245.

Lesani, M. and Palsberg, J. (2014). Decomposing opacity. In *International Symposium on Distributed Computing*, pages 391–405. Springer.

Marić, O. (2017). *Formal Verification of Fault-Tolerant Systems*. PhD thesis, ETH Zurich.

Matveev, A. and Shavit, N. (2015). Reduced hardware norec: A safe and scalable hybrid tm. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 59–71.

Pankratius, V. and Adl-Tabatabai, A.-R. (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*, pages 43–52. ACM.

Peluso, S., Palmieri, R., Romano, P., Ravindran, B., and Quaglia, F. (2015). Disjoint-access parallelism: Impossibility, possibility, and cost of TM implementations. In *Proce. of the 2015 ACM Symp. on Principles of Distri. Computing*, pages 217–226.

Peterson, C. and Dechev, D. (2017). A transactional correctness tool for abstract data types. *ACM Transactions on Architecture and Code Optimization*, 14(4):1–24.

Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10:99–116.

Wamhoff, J.-T., Riegel, T., Fetzer, C., and Felber, P. (2010). Robustm: A robust software tm. In *Symp. on Self-Stabilizing Systems*, pages 388–404.