FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 40 (2015) No. 4
ISSN 0867-6356
e-ISSN 2300-3405
DE GRUYTER OPEN
DOI: 10.1515/fcds-2015-0018

PROVING OPACITY OF TRANSACTIONAL MEMORY WITH EARLY RELEASE

Konrad SIEK [*] Paweł T. WOJCIECHOWSKI [*]

Abstract. Transactional Memory (TM) is an alternative way of synchronizing concurrent accesses to shared memory by adopting the abstraction of transactions in place of low-level mechanisms like locks and barriers. TMs usually apply optimistic concurrency control to provide a universal and easy-to-use method of maintaining correctness. However, this approach performs a high number of aborts in high contention workloads, which can adversely affect performance. Optimistic TMs can cause problems when transactions contain irrevocable operations. Hence, pessimistic TMs were proposed to solve some of these problems. However, an important way of achieving efficiency in pessimistic TMs is to use *early release*. On the other hand, early release is seemingly at odds with *opacity*, the gold standard of TM safety properties, which does not allow transactions to make their state visible until they commit. In this paper we propose a proof technique that makes it possible to demonstrate that a TM with early release can be opaque as long as it prevents inconsistent views.

Keywords: Concurrency, Parallel Programming, Software Transactional Memory, Safety, Early Release

1 Introduction

Writing parallel programs using the low-level synchronization primitives is notoriously difficult and error-prone. Over the past decade, there has been a growing interest in alternatives to lock-based synchronization by turning to the idea of software *transactional memory* (TM) [15, 22]. Basically, TM transplants the transaction abstraction from database systems and uses it to hide the details of synchronization. In particular, TM uses speculative execution to ensure that transactions in danger of reading inconsistent state abort and retry. This is a fairly universal solution and means that the programmer must only specify where transactions begin and end, and TM manages the execution so that the transactional code executes correctly and efficiently.

[*]email: `{konrad.siek,pawel.t.wojciechowski}@cs.put.edu.pl`

Thus, the programmer avoids having to solve the problem of synchronization herself, and can rely on any one of a plethora of TM systems (e.g., [5, 11, 14, 12, 18, 21]).

Since TM allows transactional code to be mixed with non-transactional code and to contain virtually any operation, rather than just reads and writes like in its database predecessors, a greater attention must be paid to the state of shared variables at any given time. For instance, if a database transaction reads a stale value, it must simply abort and retry, and no harm is done. Whereas, if a TM transaction reads a stale value it may execute an unanticipated dangerous operation, like dividing by zero, accessing an illegal memory address, or entering an infinite loop. Thus, it is insufficient for TMs to use traditional database consistency conditions like serializability [19] or strictness [3] to describe the guarantees it ensures. Instead, TMs must restrict or eliminate the ability of transactions to view inconsistent state.

To that end, the safety property called opacity [9, 10] was introduced, which includes the condition that transactions do not read values written by other live (not completed) transactions alongside serializability and real-time order conditions. Even though a number of properties then followed that tweaked its various aspects (e.g., virtual world consistency (VWC) [16], transactional memory specification (TMS1 and TMS2) [6], and markability [17]), opacity remains the gold standard of TM safety properties.

However opacity precludes *early release*, an important programming technique, where two transactions technically conflict but nevertheless both commit correctly [20], and still produce a history that is intuitively correct. This is particularly true with pessimistic concurrency control, where transactions, as a rule, do not abort. If they do not abort, then viewing the final state of a variable does not cause inconsistencies, even if the value is read from a live transaction. On the other hand systems that employ early release gain a significant improvement in performance (e.g. [23, 26]).

The contribution of this paper is to present a technique that can show that opacity can be fulfilled by transforming the original history with early release to a different form called a *decomposed history*. The transformation can be performed only under stringent assumptions with respect to the original history, but the decomposed form can then be proven to be opaque. Since we also show that the decomposed history is a *refinement* of the original history, this suffices to acknowledge that the original history provides the same safety guarantees as opacity. In this way, a TM system with early release need not necessarily relax consistency in trade for efficiency.

Our paper is composed as follows. Section 2 presents the full definition of opacity and various associated concepts. Then, Section 3 presents the intuition and formal definition behind decomposition and demonstrates the opacity of decomposed histories, and how they refine the original history. Section 4 describes an example pessimistic algorithm with early release and shows that it is opaque using decomposition. Finally, we discuss the related work in Section 5, and conclude in Section 6.

2 Definitions

Further discussion requires that we provide a definition of opacity. We begin by defining all the ancillary concepts following their original definitions from [10]. However, we adjust the wording and notation to better fit our goals in this paper. We also define the concept of observational refinement in this section, which we use in our definitions in Section 3.

2.1 Preliminaries

Let $\Pi = \{p_1, p_2, ..., p_n\}$ be a set of all processes that execute a program. Each process executes a piece of deterministic code specified by the program. Process p_k can execute local computations whose effects are not visible to other processes as well as operations on shared variables.

Variables Let $\mathbb{V}$ be a set of *shared variables* (or *variables*, in short). Each *variable*, denoted as x, y, z etc., supports *operations* that allow to retrieve or modify its state. We denote an operation on x as $op(x)w$, where value w is the argument of the operation. If the operation takes no argument, we will omit it in the notation and write $op(x)$.

Every variable x supports the following operations, denoted o, that allow to retrieve or modify its state:

a) *write* operation denoted $w(x)v$ that sets the state of x to value v; the operation's *return value* is the constant *ok*,

b) *read* operation denoted $r(x)$ that returns the current state of x.

In order to execute some operation o on variable x, process p_k issues an *invocation event* $inv^k(o)$, and receives a *response event* denoted $res^k(u)$, where u is the return value of o. The pair of these events is called a *complete operation execution* and it is denoted $o^k \to u$, whereas an invocation event $inv^k(o)$ without the corresponding response event is called a *pending operation execution.* Specifically, an execution of a read operation by process p_k is denoted $r^k(x) \to v$ and an execution of a write operation is denoted $w^k(x)v \to ok$. We refer to complete and pending operation executions as *operation executions*, denoted by *op*.

Each event is atomic and instantaneous, but the execution of the entire operation composed of two events is not.

Transactions *Transactional memory (TM)* is a programming paradigm that uses transactions to control concurrent execution of operations on shared variables by parallel processes. A *transaction* $T_i \in \mathbb{T}$ is some piece of code executed by process p_k. Hence, we say that p_k executes T_i. Process p_k can execute local computations as well as operations on shared variables as part of the transaction. In particular, the processes can execute the following operations as part of transaction T_i:

a) start_i which initializes transaction T_i, and whose return value is the constant ok_i,

b) $w_i(x)v$ and $r_i(x)$ which respectively write a value v to variable x and read x within transaction T_i, and return either the operation's return value or the constant A_i,

c) tryC_i which attempts to commit T_i and returns either the constant C_i or the constant A_i.

The constant A_i indicates that transaction T_i has been aborted, as opposed to the constant C_i which signifies a successful commitment of the transaction.

By analogy to processes executing operations on shared variables, if process p_k executes some operation as part of transaction T_i, it issues an invocation event of the form $\mathit{inv}_i^k(\mathit{start}_i)$, $\mathit{inv}_i^k(\mathit{op}_i(x)w)$ for some x, or $\mathit{inv}_i^k(\mathit{tryC}_i)$, and receives a response of the form $\mathit{res}_i^k(u_i)$. The superscript always denotes which process executes the operation, and the subscript denotes of which transaction the operation is a part. We denote operation executions by process p_k within transaction T_i as:

a) $\mathit{start}_i^k \to \mathit{ok}_i$,

b) $r_i^k(x) \to v$ or $r_i^k(x) \to A_i$,

c) $w_i^k(x)v \to \mathit{ok}_i$ or $w_i^k(x)v \to A_i$,

d) $\mathit{tryC}_i^k \to C_i$ or $\mathit{tryC}_i^k \to A_i$.

TM assumes that processes execute operations on shared variables only as part of a transaction. Furthermore, we assume that any transaction T_i is executed by exactly one process p_k and that each process executes transactions sequentially.

Even though transactions are controlled by processes, it is convenient to talk about them as separate and independent entities. Thus, rather than saying p_k executes some operation as part of transaction T_i, we will simply say that T_i performs some operation. Hence we will also forgo the distinction of processes in transactional operation executions, and write simply: $\mathit{start}_i \to \mathit{ok}_i$, $r_i(x) \to v$, $w_i(x)v \to \mathit{ok}_i$, $\mathit{tryC}_i \to C_i$, etc. By analogy, we also drop the superscript indicating processes in the notation of invocation and response events, where the distinction is not needed.

Sequential Specification Given variable x, let *sequential specification* of x, denoted $\mathit{Seq}(x)$, be a prefix-closed set of sequences containing invocation events and response events which specify the semantics of shared variables. (A set Q of sequences is *prefix-closed* if, whenever a sequence S is in Q, every prefix of S is also in Q.) Intuitively, a sequential specification enumerates all possible correct sequences of operations that can be performed on a variable in a sequential execution. Specifically, given D, the domain of variable x, and v_0 ($v_0 \in D$), an initial state of x, we denote by $\mathit{Seq}(x)$ the sequential specification of x s.t., $\mathit{Seq}(x)$ is a set of sequences of the form $[\alpha_1 \to v_1, \alpha_2 \to v_2, ..., \alpha_m \to v_m]$, where each $\alpha_j \to v_j$ ($j = 1..m$) is either:

a) $w_i(x)v_j \to \mathit{ok}_i$, where $v_j \in D$, or

b) $r_i(x) \to v_j$, and either the most recent preceding operation is $w_l(x)v_j \to \mathit{ok}_l$ ($l < i$) or there are no preceding writes and $v_j = v_0$.

From this point on we assume that the domain D of all shared variables is the set of natural numbers $\mathbb{N}_0$ and that the initial value v_0 of each variable is 0.

Histories A TM *history* H is a sequence of invocation and response events issued by the execution of transactions $\mathbb{T}_H = \{T_1, T_2, ..., T_t\}$. The occurrence and order of events in H is dictated by a given (possibly partial) execution of a program, and we treat it as if it were arbitrary. The sequence of events in history H_j can be denoted as $H_j = [e_1, e_2, ..., e_m]$. For instance, some history H_1 below is a history of a run of some program that executes transactions T_1 and T_2:

$$\begin{aligned} H_1 = [\ & inv_1(start_1), res_1(ok_1), inv_2(start_2), res_2(ok_2), \\ & inv_1(w_1(x)v), inv_2(r_2(x)), res_1(ok_1), res_2(v), \\ & inv_1(tryC_1), res_1(C_1), inv_2(tryC_2), res_2(C_2)\ ]. \end{aligned}$$

Given any history H, let $H|T_i$ be the longest subhistory (subsequence) of H that only contains invocations and responses executed by transaction T_i. For example, $H_1|T_2$ is defined as:

$$\begin{aligned} H_1|T_2 = [\ & inv_2(start_2), res_2(ok_2), inv_2(r_2(x)), res_2(v), \\ & inv_2(tryC_2), res_2(C_2)\ ]. \end{aligned}$$

Let $H|p_k$ be a subhistory of H that only contains invocations and responses executed by process p_k. We say transaction T_i *is in* H, which we denote $T_i \in H$, if $H|T_i \neq \varnothing$. Let $H|x$ be the longest subhistory of H consisting only of invocations and responses executed on variable x, but only those that form complete operation executions. A history whose all operation executions are completed is a *complete* history.

Given complete operation execution *op* that consists of an invocation event e' and a response event e'', we say *op is in H* ($op \in H$) if $e' \in H$ and $e'' \in H$. Given a pending operation execution *op* consisting of an invocation e', we say *op is in H* ($op \in H$) if $e' \in H$ and there is no other operation execution op' consisting of an invocation event e' and a response event e'' s.t. $op' \in H$.

Given two complete operation executions op' and op'' in some history H, where op' contains the response event res' and op'' contains the invocation event inv'', we say op' *precedes* op'' in H if res' precedes inv'' in H.

Most of the time it will be convenient to denote any two adjoining events in a history that represent the invocation and response of a complete execution of an operation as that operation execution, using the syntax $e \to e'$. Then, an alternative representation of $H_1|T_2$ is denoted as follows:

$$H_1|T_2 = [\ start_2 \to ok_2,\ r_2(x) \to v,\ tryC_2 \to C_2\ ].$$

History H is *well-formed* if for every transaction T_i in H, $H|T_i$ is an alternating sequence of invocations and responses s.t.,

a) starts with an invocation $inv_i(start_i)$,

b) no events in $H|T_i$ follow $res_i(C_i)$ or $res_i(A_i)$,

c) no invocation event in $H|T_i$ follows $inv_i(tryC_i)$,

d) for any two transactions T_i and T_j s.t., T_i and T_j are executed by the same process p_k, the last event of $H|T_i$ precedes the first event of $H|T_j$ in H or vice versa.

In the remainder of the paper we assume that all histories are well-formed.

History Completion Given history H and transaction T_i, T_i is *committed* if $H|T_i$ contains operation execution $tryC_i \rightarrow C_i$. Transaction T_i is *aborted* if $H|T_i$ contains response $res_i(A_i)$ to any invocation. Transaction T_i is *commit-pending* if $H|T_i$ contains invocation $tryC_i$ but it does not contain $res_i(A_i)$ nor $res_i(C_i)$. Finally, T_i is *live* if it is neither committed, aborted, nor commit-pending.

Given two histories $H' = [e'_1, e'_2, ..., e'_m]$ and $H'' = [e''_1, e''_2, ..., e''_m]$, we define their concatenation as $H' \cdot H'' = [e'_1, e'_2, ..., e'_m, e''_1, e''_2, ..., e''_m]$. We say P is a prefix of H if $H = P \cdot H'$. Then, let a *completion* $Compl(H)$ of history H be any complete history s.t., H is a prefix of $Compl(H)$ and for every transaction $T_i \in H$ subhistory $Compl(H)|T_i$ equals one of the following:

a) $H|T_i$, if T_i finished committing or aborting,

b) $H|T_i \cdot [res_i(C_i)]$, if T_i is live and contains a pending $tryC_i$,

c) $H|T_i \cdot [res_i(A_i)]$, if T_i is live and contains some pending operation,

d) $H|T_i \cdot [tryC_i \rightarrow A_i]$, if T_i is live and contains no pending operations.

Note that, if all transactions in H are committed or aborted then $Compl(H)$ and H are identical. We call any history H such that $H = Compl(H)$ a determined history.

Two histories H' and H'' are *equivalent* (denoted $H' \equiv H''$) if for every $T_i \in \mathbb{T}$ it is true that $H'|T_i = H''|T_i$. When we write H' is equivalent to H'' we mean that H' and H'' are equivalent.

Sequential and Legal Histories A *real-time order* $\prec_H$ is an order over history H s.t., given two transactions $T_i, T_j \in H$, if the last event in $H|T_i$ precedes in H the first event of $H|T_j$, then T_i *precedes* T_j in H, denoted $T_i \prec_H T_j$. We say that two transactions $T_i, T_j \in H$ are *concurrent* if neither $T_i \prec_H T_j$ nor $T_j \prec_H T_i$. We say that history H' *preserves the real-time order* of H if $\prec_H \subseteq \prec_{H'}$. A *sequential history* S is a history, s.t. no two transactions in S are concurrent in S.

We analogously define a real-time order $\prec_H$ of operation executions over history H.

Let S' be a sequential history that contains only committed transactions, with the possible exception of the last transaction, which can be aborted. We say that sequential history S' is *legal* if for every $x \in \mathbb{V}$, $S'|x \in Seq(x)$.

Using the definitions above allows us to formulate the central concept that defines consistency in opacity: *transaction legality*. Intuitively, we can say a transaction is legal in a sequential history if it only reads values of variables that were written by committed transactions or by itself. More formally, given a sequential history S and a transaction $T_i \in S$, we then say that transaction T_i is *legal in* S if $Vis(S, T_i)$ is legal,

$$\begin{array}{llllll} \{x=0\} & T_1 & [\![\ w(x)1 & &]\!] & \\ & T_2 & [\![& r(x)1 &]\!] & \{x=1\} \end{array}$$

Figure 1: Minimal example of early release representing H_{er}.

where $Vis(S, T_i)$ is the longest subhistory S' of S s.t., for every transaction $T_j \in S'$, either $i = j$ or T_j is committed in S' and $T_j \prec_S T_i$.

Unique Writes History H has *unique writes* if, given transactions T_i and T_j (where $i \neq j$ or $i = j$), for any two write operation executions $w_i(x)v' \to ok_i$ and $w_j(x)v'' \to ok_j$ it is true that $v' \neq v''$ and neither $v' = v_0$ nor $v'' = v_0$. For the remainder of the paper we focus on histories with unique writes. This assumption does not reduce generality, in that any history without unique writes trivially can be transformed into a history with unique writes (for instance, by appending a timestamp to each written value), and, as shown in [10], such a transformation does not break (final-state) opacity.

2.2 Definition of Opacity

Given the definitions presented above, opacity is defined by the following two definitions. The first definition specifies *final state opacity* that ensures the appropriate guarantees for a transactional history. The second definition uses *final state opacity* to define a safety property that is prefix closed. Both definitions follow those in [10].

Definition 1. *A finite TM history H is final-state opaque if, and only if, there exists a sequential history S equivalent to any completion of H s.t.,*

(a) S preserves the real-time order of H,

(b) every transaction T_i in S is legal in S.

Definition 2. *A TM history H is opaque if, and only if, every finite prefix of H is final-state opaque.*

2.3 Opacity and Early Release

Given the definition of opacity let us show how it does not support early release using a minimal example. This was also presented using graph representation of opacity in [25].

Formally, transaction T_i *releases x early* in H if, and only if, there is some prefix H' of H, such that T_i is live in H' and there exists T_j in H' such that there is a read operation execution $op_j \in H'|T_j$ s.t. $op_j = r_j(x) \to v'$ and a preceding write operation $op_i \in H'|T_i$ s.t. $op_i = w_i(x)v'' \to ok$ and $v' = v''$. A minimal example of this is as follows:

$$\begin{aligned} H_{er} = [\ & start_1 \to ok_1, start_2 \to ok_2, \\ & w_1(x)1 \to ok_1, r_2(x) \to 1, \\ & tryC_1 \to C_1, tryC_2 \to C_2\ ]. \end{aligned}$$

In order to aid understanding, we illustrate history H_{er} in Fig. 1 where each transaction's execution is represented in a separate line (identified as T_1 and T_2). The operation executions are ordered on the time axis in accordance to the order of operation executions in H_{er}. For the sake of brevity, we abbreviate operation execution $start_i \to ok_i$ to $\llbracket$, $r_i(x) \to v$ to $r(x)v$, $w_i(x)v \to ok_i$ to $w(x)v$, and $tryC_i \to C_i$ to $\rrbracket$. We also depict the initial and final state of variables, e.g. $\{x = 0\}$.

The example is simply a rendition of the definition of early release into a transactional history. Note that if we demonstrate that this example does not satisfy opacity, then opacity does not allow histories with early release. Below, we show that the example does not satisfy opacity.

Final-state Opacity First, let us point out that the example is final-state opaque. The completion $Compl(H_{er})$ of H_{er} is identical to H_{er}, because H_{er} contains only committed transactions (that is: for any transaction $T_i \in H_{er}$ it is true that $H_{er}|T_i = H'_i \cdot [tryC_i \to C_i]$).

Note that there exists a sequential history $S = H_{er}|T_1 \cdot H_{er}|T_2$ that is equivalent to H_{er}. Since all transactions are concurrent in H_{er}, the real time order of H_{er} is empty ($\prec_{er} = \varnothing$). Then, trivially, $\prec_{er} \subseteq \prec_S$. So S satisfies Def. 1a.

S contains operations of transactions $\mathbb{T} = \{T_1, T_2\}$ on objects $\mathbb{V} = \{x\}$. Subhistory $Vis(S, T_1) = S|T_1$ is legal since $Vis(S, T_1)|x = [w_1(x)1 \to ok_1] \in Seq(x)$ and the value 1 is in the domain of x ($\mathbb{N}_0$). Hence T_1 in S is legal in S. Subhistory $Vis(S, T_2) = S|T_1 \cdot S|T_2$ is legal as well, since $Vis(S, T_2)|x = [w_1(x)1 \to ok_1, r_2(x) \to 1]$, since 1 is in the domain of x and $r_2(x) \to 1$ is directly preceded by an operation writing 1 to x. Hence T_2 in S is legal in S.

Since every T_i in S is legal in S, then sequential history S satisfies Def. 1b. Therefore, H_{er} is final-state opaque.

Prefix-closedness Below, we show that history H_{er} is nevertheless not opaque, because, by Def. 2, opacity requires that all prefixes of a history must be final-state opaque. To that end, let history P be a prefix of H_{er} created by removing the last 4 events of H_{er}, i.e.:

$$P = [\ start_1 \to ok_1, start_2 \to ok_2, w_1(x)1 \to ok_1, r_2(x) \to 1\ ].$$

Furthermore, let $P' = Compl(P)$ s.t., $P'|T_1 = P|T_1 \cdot [tryC_1 \to A_1]$ and $P'|T_2 = P|T_2 \cdot [tryC_2 \to A_2]$. Note that, from definition of completion, P' is the only possible completion of P because only case (d) applies to both transactions in P.

There are two possible sequential histories equivalent to P'. The first one is $S' = P'|T_1 \cdot P'|T_2$. Since T_1 is aborted in P', then $Vis(S', T_2) = P'|T_2$ (that is, operation executions from $P'|T_1$ are excluded from $\mathit{Vis}(S', T_2)$). However, $\mathit{Vis}(S', T_2)$ is not legal because it contains operation execution $r_2(x) \rightarrow 1$ that is not preceded by any write and $v_0 \neq 1$. Hence T_1 in S' is not legal in S'. So S' does not bear out Def. 1b.

The second sequential history equivalent to P' is $S'' = P'|T_2 \cdot P'|T_1$. Here, $\mathit{Vis}(S'', T_2) = P'|T_2$ (because T_2 is not preceded by any other transaction in S''). Since, $\mathit{Vis}(S'', T_2) = \mathit{Vis}(S', T_2)$, then by analogy to the discussion above $\mathit{Vis}(S'', T_2)$ is not legal, so T_2 in S'' is not legal in S''. Thus, S'' does not satisfy Def. 1b either.

In effect, there is no sequential history equivalent to P' that satisfies Def. 1. Therefore, P does not satisfy Def. 1, and, since P is a prefix of H_{er}, then H_{er} does not satisfy Def. 2 and so it is not opaque.

2.4 Observational Refinement

Observational refinement [13, 2], intuitively, is a notion that given two programs and an observer who only sees the results of executing these two programs, if both programs always produce the same results, then, effectively, the programs are indistinguishable, and, therefore, interchangeable. The definition depends on what is considered observable behavior, which we assume to be the state of all variables during the execution of a program.

Given the set $\mathbb{V}$ of all variables $\mathbb{V} = \{x_1, x_2, ..., x_w\}$, let *state* $\mathcal{S}$ be a set of variables paired with their values, i.e. $\mathcal{S} = \{(x_1, v_1), (x_2, v_2), ..., (x_w, v_w)\}$. Let the *initial state* $\mathcal{S}_0$ be a state s.t., for any $x_j \in \mathbb{V}$ and $(x_j, v_j) \in \mathcal{S}_0$, $v_j = v_0$. Let $\mathcal{P}(\mathcal{S})$ be a powerset of $\mathcal{S}$ and $\mathbb{E}$ be the set of all possible invocation and response events. Then let $\mathit{eval} : \mathcal{P}(\mathcal{S}) \times \mathbb{E} \mapsto \mathcal{P}(\mathcal{S})$ be a function representing the semantics of a TM system. It is out of scope of this paper to define the complete semantics of TM, so we limit ourselves to presenting the following assumptions about *eval*.

Intuitively, we expect operations to behave deterministically based on the initial state, regardless of transaction. We also expect successful initialization and execution of commitment operations not to modify the state. More formally, given some states $\mathcal{S}$ and $\mathcal{S}'$, processes p_k and p_q variable x, value v, transactions T_i and T_j, we assume the following:

Assumption 1. *If* $(e_1, e_2) = r_i^k(x) \rightarrow v$ *and* $(e_1', e_2') = r_j^q(x) \rightarrow v$ *then* $\mathit{eval}(\mathcal{S}, e_1) = \mathit{eval}(\mathcal{S}, e_1')$ *and* $\mathit{eval}(\mathcal{S}', e_2) = \mathit{eval}(\mathcal{S}', e_2')$.

Assumption 2. *If* $(e_1, e_2) = w_i^k(x)v \rightarrow ok_i$ *and* $(e_1', e_2') = w_j^q(x)v \rightarrow ok_j$ *then* $\mathit{eval}(\mathcal{S}, e_1) = \mathit{eval}(\mathcal{S}, e_1')$ *and* $\mathit{eval}(\mathcal{S}', e_2) = \mathit{eval}(\mathcal{S}', e_2')$.

Assumption 3. *If* $(e_1, e_2) = \mathit{start}_i^k \rightarrow ok_i$ *then* $\mathit{eval}(\mathcal{S}, e_1) = \mathcal{S}$ *and* $\mathit{eval}(\mathcal{S}', e_2) = \mathcal{S}'$.

Assumption 4. *If* $(e_1, e_2) = \mathit{tryC}_i^k \rightarrow C_i$ *then* $\mathit{eval}(\mathcal{S}, e_1) = \mathcal{S}$ *and* $\mathit{eval}(\mathcal{S}', e_2) = \mathcal{S}'$.

We say that history $H = [e_1, e_2, ..., e_m]$ is *observed-state equivalent* to history $H' = [e_1', e_2', ..., e_n']$, which we denote $H \lessapprox H'$, when there exists such an injection $f : H \mapsto H'$, that for each $e_l \in H$ there exists $e_r \in H'$ s.t., if $\mathit{eval}(\mathcal{S}_{l-1}, e_l) = \mathcal{S}_l$ and

$eval(\mathcal{S}_{r-1}, e_r) = \mathcal{S}_r$, then $\mathcal{S}_{l-1} = \mathcal{S}_{r-1}$ and $\mathcal{S}_l = \mathcal{S}_r$. Furthermore, it is necessary that if $f(e_l) = e_r$ and $f(e_{l-1}) = e_q$, then $q < r$. The intuition behind this definition is that if the events in both histories were evaluated side-by-side, they would cause the same changes to the state of the system, although one of the histories would contain some operations that were not present in the other history. However, these operations would not modify the state.

We say that transactional memory system M *observationally refines* transactional memory system M' if for any history H allowed by M there exists some history H' allowed by M' s.t., $H \lessapprox H'$.

3 History Decomposition

In the previous section we showed that a history with an instance of early release cannot be opaque. However, the history shown in Fig. 1 is an intuitively correct execution. Indeed, we showed that it is final-state opaque and only its prefix is not. This intuition would be especially true if history H_1 were generated by a pessimistic TM system, where aborts do not occur. In the particular case of these systems, forming a completion by defaulting an execution of an uncommitted transaction to an abort is too conservative and leads to perfectly legal histories being unable to satisfy opacity. On the other hand, the assumption that all transactions will eventually commit is not one that can be incorporated into the definition of opacity without compromising its meaningfulness for optimistic TMs.

Therefore, rather than modifying the definition of opacity to allow for non-aborting pessimistic TMs with early release, in this section we propose a simple technique called *decomposition.* The technique allows to create a decomposed history by splitting transactions with early release into sequences of atomic single-operation transactions (given certain stringent assumptions about their execution). This history will prevent transactions with early release from violating the consistency requirements of opacity, but will nevertheless be commensurate with the original history on the basis of observational refinement.

However, please note that this is a transformation done "on paper" rather than a technique that is used during the actual execution of operations by a particular TM system, i.e., the state changes are done by the system as if all transactions were atomic.

3.1 Intuition

Intuitively, the idea behind decomposition is to redefine any non-aborting transaction that releases early as a sequence of smaller transactions, each of which performs a single complete operation (i.e., an invocation and a response) and immediately commits. Such a decomposed transaction preserves the semantics and operations of the original transaction, but, since it is executed peacemeal, it no longer meets the definition of early release. This allows a transaction that originally had early release

$$\begin{array}{lll l l}
\{x=0\} & T_1 & [\![\ r(x)0, w(x)1, r(y)0, w(y)1\ ]\!] & & \\
 & T_2 & [\![& r(x)1, w(x)2\ ]\!] & \{x=2\}
\end{array}$$

(a) History H_2 with early release.

$$\begin{array}{lllllll}
\{x=0\} & T_1 & [\![& & &]\!] & \\
 & T_{1,1} & [\![r(x)0]\!] & & & & \\
 & T_{1,2} & & [\![w(x)1]\!] & & & \\
 & T_{1,3} & & & [\![r(y)0]\!] & & \\
 & T_{1,4} & & & & [\![w(y)1]\!] & \\
 & T_2 & [\![& & r(x)1,\ w(x)2 &]\!] & \{x=2\}
\end{array}$$

(b) History H_2' that is a decomposition of history H_2.

Figure 2: Decomposition example.

to satisfy opacity.

An example of this is given in Fig. 2 where Fig. 2b shows history H_2' which decomposes H_2 from Fig. 2a. Here, T_1 from H_2 is emptied, and the original operations comprising T_1 are executed as separate sequential transactions $T_{1,1}$, $T_{1,2}$, $T_{1,3}$, and $T_{1,4}$. This can be thought of essentially as nesting transactions $T_{1,1}$, $T_{1,2}$, $T_{1,3}$, and $T_{1,4}$ within T_1 and using them to execute code within T_1.

The decomposed history may be considered interchangeable with the original history because decomposed histories behave exactly like the original histories from which they were produced. That is, the decomposed history observationally refines the original history.

3.2 Definition of Decomposition

Let H be a TM history with unique writes. Let $\mathbb{T}_{er}$ be a set of transactions s.t., $\mathbb{T}_{er} \subseteq \mathbb{T}$ and $T_i \in \mathbb{T}_{er}$ if, and only if, T_i is guaranteed to eventually commit and T_i releases some object early. We say a transaction eventually commits if the semantics of the TM ensure that it never aborts.

Given two transactions T_i and T_j ($T_i, T_j \in \mathbb{T}$) and some invocation or response event e_i executed by transaction T_i, let $\mathit{reassign}_j(e_i)$ be an event executed by T_j and defined as,

$$\mathit{reassign}_j(e_i) = \begin{cases} \mathit{inv}_j(\mathit{op}(x)w) & \text{if } e_i = \mathit{inv}_i(\mathit{op}(x)w), \\ \mathit{res}_j(u) & \text{if } e_i = \mathit{res}_i(u). \end{cases}$$

Intuitively, $\mathit{reassign}_j(e_i)$ is the same event as e_i, only executed by transaction T_j rather than T_i.

Given some transaction T_i and an event e, let $\mathit{open}_i(e)$ and $\mathit{close}_i(e)$ denote sequences defined as follows:

$$\begin{aligned} \mathit{open}_i(e) &= [\mathit{start}_i \to \mathit{ok}_i] \cdot [e], \text{ and} \\ \mathit{close}_i(e) &= [e] \cdot [\mathit{tryC}_i \to C_i]. \end{aligned}$$

Note that using *open* on the first event and *close* on the last event of some sequence of events "envelops" them in a transaction.

Then, we define *history decomposition* as follows:

Definition 3. *Given a history H, let its decomposition $H_d = Decomp(H)$ be identical to H except that every transaction $T_i \in H$ s.t., $T_i \in \mathbb{T}_{er}$ is transformed as follows:*

a) every complete operation execution in T_i that consists of an invocation event e'_i and a response event e''_i, e'_i is replaced in H_d by $open_j(reassign_j(e'_i))$ and e''_i is replaced in H_d by $close_j(reassign_j(e''_i))$, where j is fresh, i.e. there is no T_j in H,

b) every pending operation execution in T_i that only consists of an invocation event e'_i, e'_i is replaced in H_d by $open_j(reassign_j(e'_i))$, where j is fresh, i.e. there is no T_j in H.

Note that decomposition produces a set of new transactions for each transaction $T_i \in \mathbb{T}_{er}$. We call such a transaction T_i a *decomposed transaction.* The set of all transactions produced by decomposition to execute the events of a decomposed transaction T_i is denoted $\mathbb{T}^i_d$. This set explicitly contains the decomposed transaction T_i. We will refer to any transaction $T_j \in \mathbb{T}^i_d$ s.t. $T_j \neq T_i$ as a *product* of decomposition.

Next, let us show the opacity of a decomposed history, as follows.

Lemma 1. *Given H, a final-state opaque history, and $H_d = Decomp(H)$, H_d is final-state opaque.*

Proof sketch. Let S be a sequential history fulfilling Def. 1 for H. Since H is final-state opaque then, by Def. 1, every transaction T_i in S is legal in S because $Vis(S, T_i)$ is legal.

Let S_d be a sequential history identical to S except that for every event e_i in some transaction T_i in S,

a) if e_i in H was replaced in H_d by $open_j(reassign_j(e_i))$ (given some T_j), then it is also replaced in S_d by $open_j(reassign_j(e_i))$,

b) if e_i in H was replaced in H_d by $close_j(reassign_j(e_i))$ (given some T_j), then it is also replaced in S_d by $close_j(reassign_j(e_i))$,

c) otherwise it remains e_i.

In addition every decomposed transaction directly precedes all of its product transactions.

For every transaction T_k in S_d exactly one of the following is true:

a) T_k is neither a decomposed transaction nor a product transaction. In that case, if $Vis(S_d, T_k) = Vis(S, T_k)$, then, since every transaction T_k in S is legal in S because $Vis(S, T_i)$ is legal, so, by extension, T_k in S_d is legal in S_d. Alternatively, if $Vis(S_d, T_k) \neq Vis(S, T_k)$, then $Vis(S_d, T_k)$ contains operations executed by product transactions of one or more decomposed transaction T_i, where $Vis(S, T_k)$ contains operations executed by T_i. The definition of S_d implies that S_d contains the same read and write operation executions as S, but some of the operations are executed by product transactions. Since the sequential specification $\mathit{Seq}(x)$ of any variable x ignores which transaction executes the operation as long as the written and read values are correct, then, if $\mathit{Vis}(S, T_k)$ is legal, then $\mathit{Vis}(S_d, T_k)$ is also legal. Hence, T_k in S_d is legal in S_d.

b) T_k is a decomposed transaction in S_d. Then, from Def. 3, transaction T_k does not contain any read or write operation executions. Therefore, T_k in S_d is legal in S_d if there is no other transaction T_i s.t., $T_i \prec_{S_d} T_k$. This is because $\mathit{Vis}(S_d, T_k)$ contains no read or write operations, so it is in $\mathit{Seq}(x)$ for any x, and thus $\mathit{Vis}(S_d, T_k)$ is legal. Otherwise, transaction T_k is preceded in S_d by any transaction T_j ($T_j \prec_{S_d} T_k$). For the sake of simplicity, let T_j be such a transaction that there is no other transaction T_i, s.t. $T_i \prec_{S_d} T_k$ and $T_j \prec_{S_d} T_i$. Then, $\mathit{Vis}(S_d, T_k)$ contains the same read and write operation executions as $\mathit{Vis}(S_d, T_j)$. Hence, if $\mathit{Vis}(S_d, T_j)$ is legal, then $\mathit{Vis}(S_d, T_k)$ is also legal. Thus, if the preceeding transaction T_j in S_d is legal in S_d, then T_k in S_d is legal in S_d. Since we show in (a) and (c) that other types of transactions in S_d are legal in S_d and since T_k in S_d is legal in S_d if no transaction preceeds T_k, then, trivially, T_k in S_d is legal in S_d.

c) T_k is a product transaction such that $T_k \notin S$ and $T_k \in \mathbb{T}_j^d$ for some decomposed transaction T_j. In that case $\mathit{Vis}(S_d, T_k)$ is the same as $\mathit{Vis}(S_d, T_j)$ with the exception that $\mathit{Vis}(S_d, T_k)$ also contains the operations executed by product transactions $T_i \in \mathbb{T}_d^i$ of the decomposed transaction T_j, but only if $i = k$ or $T_i \prec_{S_d} T_k$. Let T_r be a special case of a product transaction of T_j for which there is no other product transaction $T_i \in \mathbb{T}_d^i$ s.t., $T_i \prec_{S_d} T_k$. Then, note that the definition of S_d implies that $S_d|T_r$ contains the same read and write operation executions as $S|T_j$, with the exception that some of the operations are executed by product transactions. By analogy to (a), since the sequential specification $\mathit{Seq}(x)$ of any variable x ignores which transaction executes the operation as long as the written and read values are correct, then, if $\mathit{Vis}(S, T_j)$ is legal, then $\mathit{Vis}(S_d, T_r)$ is also legal. Hence, T_r in S_d is legal in S_d. Since $\mathit{Seq}(x)$ is prefix closed, then if $\mathit{Vis}(S_d, T_r)$ is legal, then every prefix of $\mathit{Vis}(S_d, T_r)$ is also legal. Since T_k precedes T_r in S_d then $\mathit{Vis}(S_d, T_k)$ is a prefix of $\mathit{Vis}(S_d, T_r)$. Therefore $\mathit{Vis}(S_d, T_k)$ is legal, and thus T_k in S_d is legal in S_d.

Thus, all transactions in S_d are legal in S_d.

Trivially, S_d preserves the real-time order of H_d and $S_d \equiv H_d$.

Since S_d preserves the real-time order of H and every T_i in S_d is legal in S_d, then, by Def. 1, H_d is final-state opaque.

$\square$

Theorem 1. *Given H, a final-state opaque history, and $H_d = \mathit{Decomp}(H)$, H_d is opaque.*

Proof sketch. Since H is final-state opaque, then, from Lemma 1, $H_d = \mathit{Decomp}(H)$ is final-state opaque.

Given $H_d = \mathit{Decomp}(H)$, Def. 3 ensures that if any transaction T_i releases early in H, then it is decomposed in H_d, so all of its write operation executions are reassigned in H_d to a product transaction T_j, s.t. $T_j \in \mathbb{T}_i^d$ and each write is directly followed in H_d by a succesful commit operation executed by T_j. Consequently, if any transaction T_j writes value v to variable x, and any transaction T_k reads v from x, then T_j always commits in H_d before T_k reads v from x.

Let P be any finite prefix of H_d. The prefix potentially contains some transactions which are not completed, and therefore are aborted in the completion $P_c = \mathit{Compl}(P)$. Note, from the above, that if any transaction T_j is aborted in P_c, then there is no transaction that reads from T_j, because either any read operation reading from T_j execution would follow T_j's commit operation due to $T_j's$ decomposition, or T_j would not release early.

Hence, there can exist a sequential history $S_c \equiv P_c$ wherein any T_k in S_c is legal in S_c. Since H_d is final state opaque and P preserves the real time order of H_d, then S_c also preserves the real time order of P_c. If any a completion of any P_c is final-state opaque, then H_d is opaque. Thus, H_d is opaque. □

3.3 Observational Refinement of Decomposed Histories

Finally, we can say that the decomposition is indistinguishable from the original history, and can be used in its place for the purpose of establishing opacity, because the decomposed history reflects exactly the operations, their order, and the effect they have on the state of the system. As such, observing the execution of the original history does not differ from observing the execution of the decomposed history.

Theorem 2. *Given any complete final-state opaque history H with unique writes and its decomposed counterpart $H_d = \mathit{Decomp}(H)$, $H \lessapprox H_d$.*

Proof sketch. From Def. 3, for every event e_i in H (for any transaction T_i in H), history H_d contains either the same event e_i, or (for some T_j, s.t. $H|T_j = \varnothing$) either the sequence $\mathit{open}_j(\mathit{reassign}_j(e_i)) = [\mathit{start}_j \to \mathit{ok}_j] \cdot [\mathit{reassign}_j(e_i)]$, or the sequence $\mathit{close}_j(\mathit{reassign}_j(e_i)) = [\mathit{reassign}_j(e_i)] \cdot [\mathit{tryC}_j \to C_j]$. Note that from Assumption 3 and Assumption 4, evaluation of any event e'_j in $[\mathit{start}_j \to \mathit{ok}_j]$ or in $[\mathit{tryC}_j \to C_j]$ does not impact state, i.e., $\mathit{eval}(\mathcal{S}, e'_j) = \mathcal{S}$. Note also that Assumption 1 and Assumption 2 imply that $\mathit{eval}(\mathcal{S}, e_i) = \mathit{eval}(\mathcal{S}, \mathit{reassign}_j(e_i))$. Hence we can derive from $\mathit{Decomp}(H)$ a function $f : H \mapsto H_d$ that for any event e_i returns $\mathit{reassign}_j(e_i)$ if $\mathit{Decomp}(H)$ transforms e_i to either $\mathit{open}_j(\mathit{reassign}_j(e_i))$ or $\mathit{close}_j(\mathit{reassign}_j(e_i))$, or e_i otherwise.

Note that this function is an injection from H to H_d, and that if some event e_i is not in the range of f, it is part of a transaction initialization or transaction commitment, so $\mathit{eval}(\mathcal{S}, e_i) = \mathcal{S}$. Furthermore, note that for any event e_i in $H|T_i$ function f returns either the same event e_i or some other event e_j that represents the invocation of or response to the same operation, just executed by a different transaction T_j which is a product transaction of the decomposition of T_i. In that case, from Assumption 1 and Assumption 2 given some state $\mathcal{S}$, for any e_i it is true that $\mathit{eval}(\mathcal{S}, e_i) = \mathit{eval}(\mathcal{S}, f(e_i))$.

Finally, since function *Decomp* preserves the order of events from H in H_d, then for two events e_1 and e_2 in H, s.t. e_1 precedes e_2 in H, $f(e_1)$ precedes $f(e_2)$ in H_d. Thus, since f exists, then by definition of observed-state equivalency, $H \lessapprox H_d$. $\square$

Corollary. *Given a TM system M and a (hypothetical) TM system M_d that for every history H allowed by M produces a history $H_d = Decomp(H)$, since $H \lessapprox H_d$, then M observationally refines M_d.*

4 Safety of Pessimistic TM

In this section, as an example, we provide a proof sketch showing by decomposition that a pessimistic concurrency control algorithm that avoids conflicts and eliminates aborts is opaque despite early release.

The *Supremum Versioning Algorithm (SVA)* [28, 27] is a fully-pessimistic concurrency control algorithm. When SVA transactions start, they get a consistent snapshot composed of version numbers for each variable they will *access* (i.e. execute a read or write operation on it). Versions can be seen as permissions to access variables: they are used in conjunction with a *local counter* (also per variable) in the *access condition* to defer accesses to a given variable until preceding transactions finish operating on it. Broadly, a transaction can access a variable if it has a version number one lower than the variable's local counter. When a transaction is complete it *commits*: releases all variables by setting their local counters to its version number for the variable (this also requires meeting the access condition). To improve the parallelism of execution, transactions can release any variable they will no longer access, even before committing. This can be done automatically by the algorithm or manually by the programmer. This former is possible if a transaction knows *a priori* the upper bounds on the number of accesses to each variable (details on finding this out are given further below).

Lemma 2. *A determined finite SVA history is final-state opaque through decomposition.*

Proof sketch. Since all SVA transactions $T_i \in \mathbb{T}$ access any variable $x \in \mathbb{V}$ only when it is in the appropriate version (denoted $V_p(x, T_i)$), and since the versions of variables increase monotonically, then they have exclusive access to x. That is, an SVA transaction T_i can access x in history H only after a preceding transaction T_j releases x after last use or commits. Then, for each $x \in \mathbb{V}$, given set $\mathbb{T}_x$ that contains all transactions which access x, there exists a total order $\prec_{\mathbb{T}_x}$ on $\mathbb{T}_x$ s.t., given transactions $T_i, T_j \in \mathbb{T}_x$, $T_j \prec_{\mathbb{T}_x} T_i$ iff $V_p(x, T_j) < V_p(x, T_i)$. By extension, given any SVA history H, there exists a partial order $\preceq_H$ on H that agrees with $\prec_{\mathbb{T}_x}$ for each $x \in \mathbb{V}$ (i.e. $\prec_{\mathbb{T}_x} \subseteq \preceq_H$, for each $x \in \mathbb{V}$).

Let H be any finite SVA history that is determined (i.e. one for which $Compl(H) = H$). Let S be a sequential history equivalent to H s.t. transactions in S are ordered in accordance to $\preceq_H$. Then, trivially, S follows the real-time order of H.

Note that given a determined SVA history H, no transaction aborts in H. Note also that given two transactions $T_i, T_j \in H$, s.t. $T_i, T_j \in \mathbb{T}_x$, if T_i accesses x after T_j, then

T_i accesses x after T_j releases x or commits. Since SVA transactions release objects after last use, then any transaction always views a consistent state of the system and is the only transaction that executes operations on a given variable between its first and last access of that variable. Hence, each transaction in H behaves as if it were executed sequentially. So each transaction in any sequential history S s.t. $H \equiv S$ conforms to a sequential specification of each variable. Therefore, every transaction T_i in S is legal in S.

Since we can construct a sequential history S equivalent to H that preserves the real time order of H and every transaction T_i in S is legal in S, therefore H is final-state opaque. □

Theorem 3. *Every SVA history is opaque through decomposition.*

Proof sketch. Let H be any finite determined SVA history. Let $H_d = \mathit{Decomp}(H)$. Since H is final-state opaque (Lemma 2), then, by Theorem 1, H_d is opaque. Then, since H_d observationally refines H (Theorem 2), H is indistinguishable from an opaque history H_d.

Let H' be any SVA history that is not determined. Trivially, there exists such a determined SVA history H, that H' is a prefix for H. Since every every determined history is final state opaque (Lemma 2), there exists a decomposed history $H_d = \mathit{Decomp}(H)$ that is opaque (Theorem 1). Then, there must exist a $H'_d = \mathit{Decomp}(H')$ that is a prefix of H_d. Since all prefixes of H_d are final-state opaque, then H'_d is final-state opaque (Def. 2). Also, since all prefixes of H_d are final-state opaque, then all prefixes of H'_d are final-state opaque, and in consequence H'_d is opaque. Then, since H'_d observationally refines H' (Theorem 2), H' is indistinguishable from an opaque history. □

5 Related work

Opacity [9, 10] can be considered the standard TM safety property, but it does not take into account the possibility of a completely abort-free TM which can release early without violating consistency (as we describe in Section 2). To the best of our knowledge we are the first to propose reconciling such TMs with opacity.

Other work has been done to weaken various aspect of opacity. Among these, *virtual world consistency* [16] allows aborted transactions not to agree on the witness history and *virtual time opacity* [16] relaxes the real-time order condition. *View transactions* [1] only require that a transaction commits on any snapshot, that can be different than the one the transaction operated on, but if the transaction did operate on it, the results would be externally indistinguishable. *Elastic opacity* [8] allows transactions to roll back partially in case of conflicts.

In addition some properties were introduced specifically to incorporate early release for TM models with potentially aborting transactions. This includes our work on *last-use opacity* [24] where transactions can only release early if the value of the variable in question will remain constant after release. Another example are a family of *live* properties, including recent work on *live opacity* [7], which allow transactions to release early given that the transaction is guaranteed to commit afterward (although in general transactions can abort).

6 Conclusions

The paper presents a technique called decomposition that aids in proving opacity for TM systems with guaranteed commitment where transactional histories can contain early release. The technique splits transactions with early release into smaller atomic units. We show that if the original history is final-state opaque, the decomposed history is opaque. We also demonstrate that a final-state opaque history whose decomposition is opaque may be considered opaque due to observational refinement. In addition, we show an example of the application of the technique for SVA, a pessimistic TM concurrency control algorithm with early release which completely prevents aborts.

TM systems gain a significant performance boost from the application of early release (see e.g. [14, 20, 8, 4, 23, 26]), but systems with early release cannot satisfy the definition of opacity, even if they produce no histories with inconsistent views. Our proposition to show that histories with early release can be seen as indistinguishable from opaque histories via the application of decomposition means that safety is not traded for efficiency in TM systems with early release if transactions that release early never abort.

While the technique should allow to build proofs for pessimistic TM with guaranteed commitment, it would be interesting to devise methods or properties for demonstrating the correctness guarantees of TMs with both early release and rollback capability.

Acknowledgements The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

References

[1] Afek, Y., Morrison, A., and Tzafrir, M. Brief announcement: View Transactions: Transactional Model with Relaxed Consistency Checks. In *Proc. PODC'10*, 2010.

[2] Attiya, H., Gotsman, A., Hans, S., and Rinetzky, N. A programming language perspective on transactional memory consistency. In *Proc. PODC'13*, 2013.

[3] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[4] Bieniusa, A., Middelkoop, A., and Thiemann, P. Brief announcement: actions in the twilight—concurrent irrevocable transactions and inconsistency repair. In *PODC '10*, 2010.

[5] Dice, D., Shalev, O., and Shavit, N. Transactional Locking II. In *Proc. DISC'06*, 2006.

[6] Doherty, S., Groves, L., Luchangco, V., and Moir, M. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25, Sept. 2013.

[7] Dziuma, D., Fatourou, P., and Kanellou, E. Consistency for transactional memory computing. *Bulletin of the EATCS*, 113, 2014.

[8] Felber, P., Gramoli, V., and Guerraoui, R. Elastic Transactions. In *Proc. DISC'09*, Sept. 2009.

[9] Guerraoui, R. and Kapałka, M. On the correctness of transactional memory. In *Proc. PPoPP'08*, Feb. 2008.

[10] Guerraoui, R. and Kapałka, M. *Principles of Transactional Memory.* Morgan & Claypool, 2010.

[11] Harris, T. and Fraser, K. Language Support for Lightweight Transactions. In *Proc. OOPSLA'03*, Oct. 2003.

[12] Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. Composable memory transactions. In *Proc. PPoPP'05*, June 2005.

[13] He, J., Hoare, C. A. R., and Sanders, J. W. Data refinement refined. In *Proc. ESOP'86*, 1986.

[14] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *Proc. PODC'03*, 2003.

[15] Herlihy, M. and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA'93*, May 1993.

[16] Imbs, D., de Mendivil, J. R., and Raynal, M. On the Consistency Conditions or Transactional Memories. Technical report, Dec. 2008.

[17] Lesani, M. and Palsberg, J. Decomposing opacity. In *Proc. DISC'14*, Oct. 2014.

[18] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. Design and implementation of transactional constructs for C/C++. In *Proc. OOPSLA'08*, 2008.

[19] Papadimitrou, C. H. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

[20] Ramadan, H. E., Roy, I., Herlihy, M., and Witchel, E. Committing Conflicting Transactions in an STM. In *Proc. PPoPP'09*, Feb. 2009.

[21] Ringenburg, M. F. and Grossman, D. AtomCaml: first-class atomicity via roll-back. In *Proc. ICFP'05*, Sept. 2005.

[22] Shavit, N. and Touitou, D. Software Transactional Memory. In *Proc. PODC'95*, Aug. 1995.

[23] Siek, K. and Wojciechowski, P. T. Atomic RMI: a Distributed Transactional Memory Framework. In *Proc. HLPP'14*, July 2014.

[24] Siek, K. and Wojciechowski, P. T. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proc. DISC'14*, 2014.

[25] Siek, K. and Wojciechowski, P. T. Zen and the Art of Concurrency Control: An Exploration of TM Safety Property Space with Early Release in Mind. In *Proc. WTTM'14*, July 2014.

[26] Siek, K. and Wojciechowski, P. T. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming*, 2015.

[27] Wojciechowski, P. T. Isolation-only Transactions by Typing and Versioning. In *Proc. PPDP'05*, July 2005.

[28] Wojciechowski, P. T., Rütti, O., and Schiper, A. SAMOA: A Framework for a Synchronisation–Augmented Microprotocol Approach. In *Proc. IPDPS'04*, Apr. 2004.

Received 24.03.2015, accepted 25.08.2015