Decentralized Runtime Analysis of Multithreaded Applications

Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu
Department of Computer Science, University of Illinois at Urbana Champaign
{ksen,vardhan,agha,grosu}@cs.uiuc.edu

Abstract

Violations of a number of common safety properties of multithreaded programs–such as atomicity and absence of dataraces–cannot be observed by looking at the linear execution trace. We characterize a class of such properties, called robust properties, and define a simple but expressive epistemic logic to specify them. We then develop an efficient algorithm to automatically monitor and predict violations of robust safety properties. Our algorithm is based on capturing the causal structure of a computation through a mechanism similar to vector clock updates. The algorithm automatically synthesizes decentralized monitors to evaluate the information at each thread and to detect and predict safety violations. Based on this approach, a tool named DAME *has been developed and evaluated on some simple examples.*

1. Introduction

In multithreaded systems, multiple threads execute concurrently and communicate with each other by reading or writing to shared variables. Moreover, the threads synchronize by acquiring and releasing locks. The concurrent execution of threads results in different possible interleavings between the threads and this nondeterminism often results in errors in deployed systems that were not observed during the software testing cycle. The nondeterminism has another consequence: it is not feasible to recreate what led to the error by simply re-executing the program on the same inputs.

One approach to this problem is to use heavy-weight formal techniques, such as theorem proving or model-checking, which account for all possible executions. Unfortunately, despite impressive gains in these techniques, it is still not feasible to rigorously verify large-scale systems. This has led to an interest in light-weight techniques to improve the testing of large software systems and allow the monitoring of their runtime behavior. However, it has been previously observed that the naïve monitoring of the linear execution of multithreaded programs against safety requirements traces masks bugs that could have been detected using smarter runtime analysis techniques.

To address this problem, we proposed a predictive analysis technique and implemented it in a tool called JMPaX [24, 25]. JMPaX predicts latent errors that are not apparent in a successful execution of a multithreaded program. Unfortunately, JMPaX involves considerable overhead: events are collected centrally in order to construct a partial *computation lattice*. In the present work, we distribute the task of monitoring a computation among the threads in order to predict bugs. One advantage of the current approach is that as soon as the monitoring technique catches a potential safety violation, it may invoke recovery code to bring the system back to a safe state by, for example, rebooting the system, or releasing resources.

We are interested in properties such as atomicity and dataraces. Note that it is not possible to observe such properties from a linear trace of a system since such traces do not provide information about the causal structure of a computation.

We restrict consideration to a class of properties we call *robust properties*: a possible interleaving of a concurrent program satisfies a robust property if and only if all *causally consistent* interleavings satisfy the property. Examples of robust properties include absence of datarace on some variable, and atomic execution of a block of code in a single thread. We define a novel logic which enables us to represent robust properties and develop an algorithm which allows us to detect violations of these properties. We demonstrate the technique by means of some examples executed using the tool and show results which suggest that the execution may be relatively efficient.

The work described in this paper makes three significant contributions. First, we define a simple but expressive logic to specify safety properties in multithreaded systems and show that the logic is able to state many properties that are of interest in multithreaded systems. Second, we provide an algorithm to synthesize decentralized monitors for safety properties expressed in this logic. Finally, we describe the results of an implementation of a tool, DAME, that is based on our technique. The tool is publicly available for download [2].

2. Decentralized Analysis of Multithreaded Systems

In this section we present the machinery we use. First, we formalize the notion of causality in multithreaded systems. We then introduce multithreaded temporal logic (MtTL) as an appropriate underlying formalism for expressing causal safety requirements of multithreaded systems. Finally, we present an automatic decentralized monitor generation algorithm for MtTL requirements.

2.1. Multithreaded Executions and Causality

A multithreaded program consists of n threads t_1, t_2, ..., t_n that execute concurrently and communicate with each other through a set of shared variables. The computation of each thread is abstracted out in terms of *events*, while the multithreaded computation is abstracted out in terms of a partial order $\prec$ on events. There can be three types of events: an *internal* event, a *read* or a *write* of a shared variable. Internal events can be reads or writes of local variables. We use e_i^j to represent the j^{th} event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e, e', etc.; we may write $e \in t_i$ when event e is generated by thread t_i. Let us fix an arbitrary but fixed multithreaded execution and let S be the set of all variables that were shared by more than one thread in the execution. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e', written $e \lessdot_x e'$, iff e and e' are variable access events (reads or writes) to the same variable x by different threads, and e "happens before" e', that is, e is an access to the variable x that occurred in the multithreaded execution some time before e'. This can be realized via a counter for each shared variable, which is incremented at each variable access.

Let E_i denote the set of events of thread t_i and let E denote $\bigcup_i E_i$. Also, let $\lessdot \subseteq E \times E$ be defined as follows:

1. $e \lessdot e'$ whenever e and e' are events of *the same thread* and e happens before e';
2. $e \lessdot e'$ whenever there is an $x \in S$ with $e \lessdot_x e'$ and at least one of e, e' is a write.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. This partial order captures the *happens-before* relation among the events in different threads. The structure described by $\mathcal{C} = (E, \prec)$ is called a *multithreaded computation*. Let us define $\preccurlyeq$ as the reflexive and transitive closure of $\lessdot$. A permutation of all events in E that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with $\prec$, is called a *consistent multithreaded run*, or simply, a *multithreaded run*. Any such consistent run can be a valid execution of the multithreaded program.

The definition of the happens-before relation above may look technically straightforward; however, its significance with respect to accessing and passing information in a multithreaded system should not be underestimated. Because of its crucial role in our approach to decentralized analysis and in particular in the rest of the paper, we next briefly discuss it informally, from a generic information-propagation perspective. Multithreaded systems are routinely regarded as distributed systems in which communication is realized via shared memory. Let us regard threads as agents performing their individual tasks, accumulating and passing information about the state of the system when (and only when) they access the shared variables – the only communication "media" available. What "information" is and how it can be formalized depends upon the particular application, so we deliberately let it vague for the time being, mentioning only that it can refer to the entire execution history of the system; Section 2.3 shows a concrete formalization of information as "knowledge", in the context of monitoring safety requirements expressed as decentralized temporal properties. Shared variables can also be regarded as information-bearing agents, carrying not only corresponding values, but also information about the system, accumulated from interaction with other thread agents. Before we discuss how information is propagated into the system, note that the intuitive meaning of an event e of thread t "happened-before" event e' of thread t', that is, $e \prec e'$, is that *t' had more information when e' took place than t had when e took place.*

We divide the information a shared variable has into public and private information in order to capture the causal structure of a computation. Each time a variable is read by a thread, the variable may acquire some information about the global state from the thread, but this information does not affect the causal flow of a computation. The causal structure of a computation is affected only by writes: for example, if two consecutive reads of the same variable are shuffled, the shuffle cannot change any observable properties of the computation. On the other hand, a write cannot be shuffled with any other access to a variable. This means that the state of the computation during all past reads can-

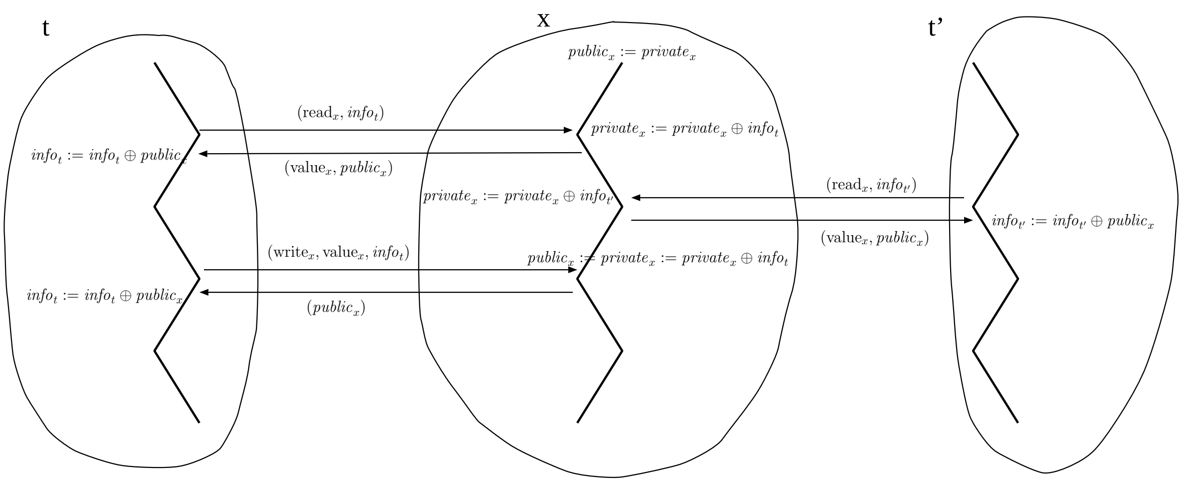

Figure 1. Multithreaded executions propagate information. Shared variable accesses (transactions) carry and update ($\oplus$) information. Each variable x has two types of information: public ($public_x$) and private ($private_x$) information. Each thread t has only one type of information ($info_t$). Reads/writes access public information, writes also update it.

not be shuffled with anything after a write. For each variable, we store the information that the variable acquires from reads as private information. To capture the causal structure, we require that each time a variable is written to, the variable discharge all information that it has acquired about the global state from past reads. On the other hand, whenever a thread reads a variable, the thread receives only the information that the variable has acquired up to the previous write to the variable. Whenever a write is performed, the information discharged now becomes the variable's public information which will now be passed on for all reads, until there is another write.

Figure 1 shows how the information is propagated and updated in a multithreaded execution, where the symbol $\oplus$ is used for information updates; for example, $public_x := private_x := private_x \oplus info_t$ should be read as follows: the private information of x is first updated with the information of thread t that was passed together with its write transaction; then the public information of x is set to its private information. Note that there is no exchange of information between the two reads initiated by the two threads; the threads only access the public information of x and update their own information. Once the write event of t takes place, t will be passed all the private information of x, which includes all the information that t' had when it read x.

For $e \in E$ we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \preccurlyeq e\}$ as the set of events that happened before e according to the $\preccurlyeq$ relation. For $e \in E_i$, we can think of $\downarrow e$ as an abstract state of the multithreaded system that the thread t_i *is aware of*, namely the state obtained after all the events in $\downarrow e$ (and only the events in $\downarrow e$) have occurred. From an information-propagation perspective, t_i simply lacks information on whether any of the other threads in the system have ever advanced after their last event that t_i was aware of. Therefore, the state after all the events in $\downarrow e$ has happened is t_i's safest guess about the current state of the system, which is why we call it *t_i's predicted global state after event e*, or simply the *predicted state* if t_i and e are understood. We will also use $\downarrow e$ to represent the predicted state. It is important to observe that this predicted state may have *not* been actually exhibited by the actual multithreaded execution. However, it may certainly be exhibited by another possible execution under different running speeds of threads and, moreover, that potential execution can be quickly inferred from the actual execution with no semantic knowledge about the program.

It is important to note that the predicted state of t_i after the event e_i is *the same* in all consistent runs. Therefore, the notion of predicted state of a thread is well defined for multithreaded computations $(E, \prec)$. We extend the definition of $\lessdot$, $\prec$ and $\preccurlyeq$ to predicted states such that $\downarrow e \lessdot \downarrow e'$ iff $e \lessdot e'$, $\downarrow e \prec \downarrow e'$ iff $e \prec e'$, and $\downarrow e \preccurlyeq \downarrow e'$ iff $e \preccurlyeq e'$. We denote the set of predicted states of a thread t_i by $PS_i \stackrel{\text{def}}{=} \{\downarrow e \mid e \in E_i\}$ and let $PS \stackrel{\text{def}}{=} \bigcup_i PS_i$. We use the symbols s_i, s_i', s_i'' and so on to represent the predicted states with respect to thread t_i. We use the notation $@_j(s_i)$ to refer to the *latest predicted state of* t_j *that* t_i *was aware of when it was in state* s_i. Formally, if $@_j(s_i) = s_j$ then $s_j \in PS_j$, $s_j \preccurlyeq s_i$, and for all $s'_j \in PS_j$ if $s'_j \preccurlyeq s_i$ then $s'_j \preccurlyeq s_j$.

For example, let us consider the set of statements executed by three threads t_1, t_2, and t_3 in Figure 2. Assuming that initially $\{x = 0, y = 0, u = 0, v = 0\}$, the predicted states for some interesting events e_3^5, e_2^4 and e_2^7 are respectively: $\{x = 1, y = 1, u = 3, v = 1\}$, $\{x = 1, y = 1, u = 3, v = 4\}$, and $\{x = 1, y = 1, u = 3, v = 5\}$. Note that the actual global state of the program in this particular execution at the time event e_2^7 occurred is $\{x = 1, y = 1, u = 9, v = 5\}$ which is different from the predicted state for event e_2^7.

2.2. Multithreaded Temporal Logic

Safety properties are routinely [16] expressed as temporal formulae of the form "always φ", where φ is a *past-time* formula. Once violated, a safety formula can never be satisfied in the future. In the context of monitoring or on-line analysis of safety properties, one needs to simply "monitor φ". Therefore, the future temporal operator "always" is replaced by "monitor", which justifies the terminology "monitoring past-time temporal properties" that is often used as an equivalent to monitoring safety [13]. Following [26], one of the major claims of our decentralized runtime analysis approach proposed in this paper is that the common past-time temporal logics are *insufficient* for stating many interesting safety properties of multithreaded systems. This claim is backed not only by all the examples in this paper, which would be hard or impossible to specify using, for example, linear temporal logic, but also by the important observation that the common temporal logics are not *multithreaded-computation invariant*, i.e., there may be a multithreaded execution which satisfies a formula φ while another execution *which is consistent with it* does not satisfy φ. There is a third, more philosophical reason for which common temporal logics are, in our view, not always appropriate to express properties to be monitored on multi-threaded systems: they assume that the atomic propositions (i.e., those involving no temporal operators) can be evaluated at any moment on the global state of the system, thus imposing a centralized view of the system in which the monitor acts as an omnipresent observer. While this can be technically accomplished on today's platforms, such as within the JVM, it is generally believed that omnipresent observers of concurrent systems have more of a theoretical relevance than practical, because they would add a prohibitive runtime overhead to the monitored systems.

Inspired by [26], we next introduce a temporal logic which we found quite expressive to state properties of multithreaded systems. The distinctive feature of this logic is that it is *decentralized* in nature: its semantics is given over multithreaded computations, with no reference whatsoever to the actual absolute global state of the system but only to states that individual threads *know about* (or *predict*) as a result of information propagation through shared variables (as explained in Section 2.1). We call this logic *multithreaded temporal logic* and abbreviate it MtTL. Since in this paper our focus is on monitoring safety properties, we only discuss the past-time fragment of this logic. MtTL extends past-time Linear Temporal Logic [16] with the powerful *epistemic operator* [21], written @, whose role is to evaluate an expression or a formula referring to a thread at a different thread; more precisely, if $@_j(\beta)$ occurs as part of a formula or expression at thread t_i currently in state s_i, then its result will be the evaluation of β in *the latest known* (at t_i) state of thread t_j, that is, in $@_j(s_i)$. We call such an expression or a formula *remote*. A remote expression or formula, refers to the *predicted state* of the remote thread, can contain nested epistemic operators. By allowing remote expressions in addition to remote formulae, we allow one to specify a larger class of desirable properties of multithreaded programs without sacrificing the efficiency of monitoring.

Consider, for example, the simple property at a thread t_i: if α is true in the current state (say s_i) of t_i then β must be true at the latest state of thread t_j which happens before s_i, that is, at $@_j(s_i)$. This will be written formally in MtTL as $@_i(\alpha \rightarrow @_j\beta)$. However, referring to *remote formulae* only is *not* sufficient in order to express a broad range of useful properties such as "at thread t_i, the value of x in the current (predicted) state is greater than the value of y in the latest known (predicted) state of thread t_j". Therefore, one would like to also be able to refer to *remote expressions*. For example, the property above can be formally specified as the MtTL formula $@_i(x > @_j y)$. Here $@_j y$ is the value of y in the latest (predicted) state of thread t_j that thread t_i is aware of.

The intuition underlying MtTL is that a multithreaded system is associated with a *global specification*, consisting of a property given as an MtTL *formula* per thread which, due to the epistemic operators, can refer to properties in the latest known states of other threads. A multithreaded computation satisfies such a specification iff each local property is satisfied by the predicted states of the corresponding thread. Next we present MtTL formally.

Syntax. An MtTL formula within the scope of an $@_i$ operator is called a *i-formula*. We let F_i, F'_i, etc., denote i-formulae. Additionally, we introduce the no-

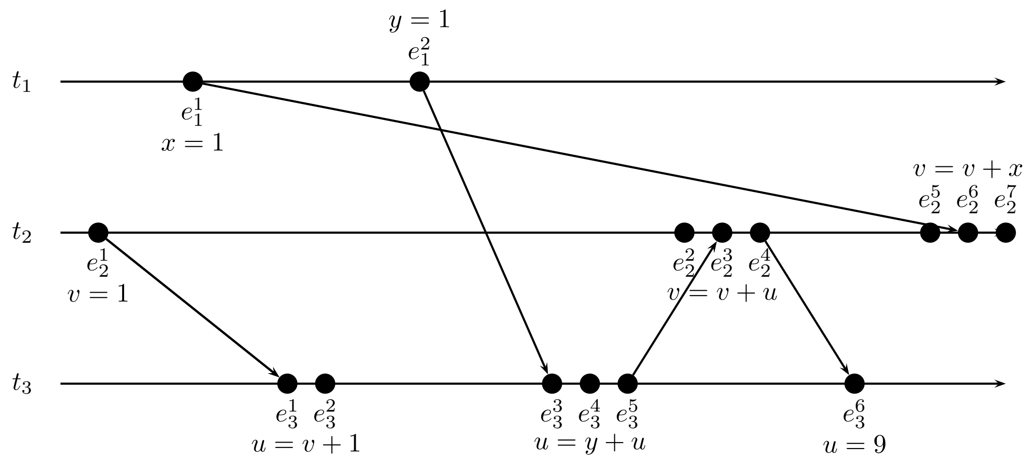

Figure 2. Example multithreaded execution. A single line of code can result in many events, for example, $u = y + u$ results in : e_3^3 (read(y)), e_3^4 (read(u)) and e_3^5 (write(u))

tion of expression local to a thread t_i, or *i-expressions*, and let ξ_i, ξ_i', etc., denote these. Informally, an i-expression is an expression over the predicted state of thread t_i and the latest known predicted states of other threads. Local predicates on i-expressions form the atomic propositions on which the temporal i-formulae are built. We add the *epistemic operators* $@_j$ that take j-expressions or j-formulae and convert them into expressions or formulae local to thread t_i. Informally, $@_j$ yields an expression or a formula over the latest known state of thread t_j at thread t_i. The following gives the formal syntax of MTTL where i and j are any thread indexes (not necessarily distinct):

$$\begin{array}{lll}
F ::= & @_i F_i & \\
F_i ::= & \text{true} \mid \text{false} \mid P(\vec{\xi_i}) \mid \neg F_i \mid F_i \ op \ F_i & \text{propositional} \\
 & \mid \odot F_i \mid \Diamond F_i \mid \boxdot F_i \mid F_i \ \S \ F_i & \text{temporal} \\
 & \mid @_j F_j & \text{epistemic} \\
\xi_i ::= & c \mid v_i \mid f(\vec{\xi_i}) & \text{functional} \\
 & \mid @_j \xi_j & \text{epistemic} \\
\vec{\xi_i} ::= & (\xi_i, \ldots, \xi_i) &
\end{array}$$

A top-level MTTL formula F is always of the form $@_i F_i$ implying that it is always specified local to a thread. The infix operator *op* can be any binary propositional operator such as $\wedge, \vee, \rightarrow, \equiv$. The term $\vec{\xi_i}$ stands for a tuple of expressions at thread t_i. The term $P(\vec{\xi_i})$ is a (computable) predicate over the tuple $\vec{\xi_i}$ and $f(\vec{\xi_i})$ is a (computable) function over the tuple. For example, P can be $<, \leq, >, \geq, =$. Similarly, some examples of f are $+, -, /, *$. Variables v_i belong to a set V_i containing all the local state variables of thread t_i. Constants such as $0, 1, 3.4$ are represented by c. The expression $@_j \xi_j$ is an i-expression representing the remote expression ξ_j. Similarly, $@_j F_j$ is an i-formula referring to the local knowledge about the remote validity of j-formula F_j. In other words, $@_j$ converts a j-expression or a j-formula to an i-expression or an i-formula, respectively.

Semantics. The semantics of MTTL is a natural epistemic extension of past-time linear temporal logic. The atomic propositions of linear temporal logic are replaced by predicates over tuples of expressions. Table 1 formally gives the semantics of each operator of MTTL. $(\mathcal{C}, s_i)[\![@_j \xi_j]\!]$ is the value of the expression ξ_j in the state $s_j = @_j(s_i)$ which is the latest known state of t_j at state s_i of thread t_i. We assume that expressions are properly typed. Typically these types would be: `integer, real, strings` etc. We also assume that $s_i, s_i', s_i'', \ldots \in PS_i$ and $s_j, s_j', s_j'', \ldots \in PS_j$. Notice that, like in the past-time linear temporal logic in [13], the meaning of the "previously" operator on the initial state of each process reflects the intuition that the execution trace is unbounded in the past and stationary.

2.3. Decentralized Monitoring Algorithm

We next describe an automated technique to synthesize efficient decentralized monitors for safety properties of multithreaded programs expressed in MTTL. We assume that one or more threads are associated with MTTL formulae which must be satisfied by the multithreaded computation. The synthesized monitor is *decentralized*, in the sense that it consists of separate, *local monitors* running on each thread. The key guiding principles in the design of this technique are: *(a)* the local monitors should be fast, so that monitoring can be done online; *(b)* the local monitors should have

$$
\begin{array}{lll}
\mathcal{C}, s \models\ @_i F_i & \text{iff } \mathcal{C}, s_i \models\ F_i \text{ where } s_i \text{ is the predicted state of } t_i & \\
 & \text{when the program is in state } s & \\
\mathcal{C}, s_i \models\ \text{true} & \text{for all } s_i & \mathcal{C}, s_i \not\models\ \text{false for all } s_i \\
\mathcal{C}, s_i \models\ P(\xi_i, \ldots, \xi_i') & \text{iff } P((\mathcal{C}, s_i)\llbracket \xi_i \rrbracket, \ldots, (\mathcal{C}, s_i)\llbracket \xi_i' \rrbracket) = \text{true} & \\
\mathcal{C}, s_i \models\ \neg F_i & \text{iff } \mathcal{C}, s_i \not\models\ F_i & \\
\mathcal{C}, s_i \models\ F_i\ op\ F_i' & \text{iff } \mathcal{C}, s_i \models\ F_i \text{ op } \mathcal{C}, s_i \models\ F_i' & \\
\mathcal{C}, s_i \models\ \odot F_i & \text{iff if } \exists s_i' \,.\, s_i' \lessdot s_i \text{ then } \mathcal{C}, s_i' \models\ F_i \text{ else } \mathcal{C}, s_i \models\ F_i & \\
\mathcal{C}, s_i \models\ \Diamond\!\!\!\cdot\;\, F_i & \text{iff } \exists s_i' \,.\, s_i' \preccurlyeq s_i \text{ and } \mathcal{C}, s_i' \models\ F_i & \\
\mathcal{C}, s_i \models\ \boxdot F_i & \text{iff } \mathcal{C}, s_i \models\ F_i \text{ for all } s_i' \preccurlyeq s_i & \\
\mathcal{C}, s_i \models\ F_i\ \S\ F_i' & \text{if } \exists s_i' \,.\, s_i' \preccurlyeq s_i \text{ and } \mathcal{C}, s_i' \models\ F_i' \text{ and } \forall s_i'' \,.\, s_i' \prec s_i'' \preccurlyeq s_i \text{ implies } \mathcal{C}, s_i'' \models\ F_i & \\
\mathcal{C}, s_i \models\ @_j F_j & \text{iff } \mathcal{C}, s_j \models\ F_j \text{ where } s_j = @_j(s_i) & \\
 & & \\
(\mathcal{C}, s_i)\llbracket v_i \rrbracket & = s_i(v_i), \text{ that is, the value of } v_i \text{ in } s_i & \\
(\mathcal{C}, s_i)\llbracket c_i \rrbracket & = c_i & \\
(\mathcal{C}, s_i)\llbracket f(\xi_i, \ldots, \xi_i') \rrbracket & = f((\mathcal{C}, s_i)\llbracket \xi_i \rrbracket, \ldots, (\mathcal{C}, s_i)\llbracket \xi_i' \rrbracket) & \\
(\mathcal{C}, s_i)\llbracket @_j \xi_j \rrbracket & = (\mathcal{C}, s_j)\llbracket \xi_j \rrbracket \text{ where } s_j = @_j(s_i) &
\end{array}
$$

Table 1. Semantics of MTTL

little memory overhead, in particular, should *not* store the entire history of events on a process; *(c)* the local monitors should not change the semantics of the original multithreaded program. In what follows, by remote expression or formula we mean one which occurs in any of the monitored MTTL formulae.

Decentralized monitoring is efficiently performed by maintaining instances of a data-structure, called *knowledge vector*, with every thread and with every shared variable. A knowledge vector is an array with an entry for any thread t_j for which there is an occurrence of $@_j$ in any MTTL formula at any thread and an entry for every shared variable that appears in any MTTL formula. Knowledge vectors compactly encode an instance of the generic notion of "information" passed in a multithreaded computation (see Section 2.1), and are motivated and inspired by vector clocks [10, 17] and especially multithreaded vector clocks [25]. The total size of a knowledge vector is *not* dependent on the number of threads, but on the number of remote expressions/formulae and the number of relevant shared program variables. A knowledge vector KV may contain an entry for thread t_j, denoted by $KV[j]$ or an entry for a shared variable x, denoted by $KV[x]$. If the entry corresponds to thread t_j then $KV[j]$ contains the following fields:

- $KV[j].seq$: sequence number of the latest event at t_j;
- $KV[j].val$: set of values of j-expressions/formulae.

If the entry corresponds to a variable x then $KV[x]$ contains:

- $KV[x].seq$: number of writes to the variable x;
- $KV[x].val$: the value of x.

Each thread t_i keeps a local knowledge vector denoted by KV_i. Each shared variable x maintains two knowledge vectors denoted by KV_x^a (access) and KV_x^w (write). Based on the informal intuition about information propagation we provided in Section 2.1, KV_i stores the information of thread i, while KV_x^a and KV_x^w store the private and the public information of x, respectively. The information encoded in the knowledge vectors is updated entry-wise. If $KV[l]$ and $KV'[l]$ are entries in two knowledge vectors for some l which is either a thread identifier or a shared variable, then we let $\max(KV[l], KV'[l])$ be the entry having the most recent information, that is, the one having the largest sequence number: if $KV[l].seq > KV'[l].seq$ then it is $KV[l]$, otherwise it is $KV'[l]$. Whenever thread t_i with knowledge vector KV_i processes the event e_i^k, the following algorithm is executed:

(1) If e_i^k is **internal** (read/write of local variable), then

- $KV_i[i].seq \leftarrow KV_i[i].seq + 1$
- Update $KV_i[i]$ in the predicted state of t_i after the event e_i^k, say s_i. For this, we evaluate all expressions ξ_i and all formulae F_i that occur in any MTTL formula as remote (i.e., as $@_i\xi_i$ and $@_iF_i$) using a simple recursive implementation of the semantics in Table 1 that needs to only store one bit per local temporal operator (similar to the one presented in detail in [13, 26]), and then store them in $KV_i[i].val$. In case any of these ξ_i or F_i refer to remote expressions or formulae, say of a thread j, then they only need to lookup into $KV_i[j].val$ for their most recent known values.

(2) If e_i^k is a **read** of a shared variable x, then

- $KV_i[i].seq \leftarrow KV_i[i].seq + 1$
- For all l (thread or variable) different from i do
  $KV_i[l] \leftarrow \max(KV_i[l], KV_x^w[l])$
  $KV_x^a[l] \leftarrow \max(KV_x^a[l], KV_i[l])$
- Update $KV_i[i]$ as in the second bullet in **(1)** above
- $KV_x^a[i] \leftarrow KV_i[i]$;

(3) If e_i^k is a **write** of shared variable x (say, with value v), then

- $KV_i[i].seq \leftarrow KV_i[i].seq + 1$
- For all l (thread or variable) different from i do $KV_x^a[l] \leftarrow KV_x^w[l] \leftarrow KV_i[l] \leftarrow \max(KV_x^a[l], KV_i[l])$
- If $KV_i[x]$ exists then
  $KV_x^a[x].seq \leftarrow KV_x^w[x].seq \leftarrow KV_i[x].seq \leftarrow KV_i[x] + 1$
  $KV_x^a[x].val \leftarrow KV_x^w[x].val \leftarrow KV_i[x].val \leftarrow v$
- Update $KV_i[i]$ as in the second bullet in **(1)** above
- $KV_x^a[i] \leftarrow KV_x^w[i] \leftarrow KV_i[i]$.

We call this the KNOWLEDGEVECTOR algorithm. Informally, $KV_i[j].val$ contains the latest values that t_i has for j-expressions or j-formulae. Therefore, for the value of a remote expression or formula of the form $@_j \xi_j$ or $@_j F_j$, process p_i can just use the entry corresponding to ξ_j or F_j in the set $KV_i[j].val$. The correctness of this algorithm is given by the following results:

Lemma 1 *For any x for which $KV_i[x]$ is defined, $KV_i[x].val$ contains the value of x in the current predicted state of t_i.*

Proof in appendix.

Lemma 2 *For any thread t_i and any j, the entry for ξ_j or F_j in $KV_i[j].val$ contains the value of $@_j \xi_j$ or $@_j F_j$, respectively.*

Proof follows from [22]

Theorem 3 *For any* MTTL *formula $@_i F_i$, $\mathcal{C}, s \models @_i F_i$ (or $\mathcal{C}, s \not\models @_i F_i$) if and only if the value of $@_i F_i$ in $KV_i[i].val$ in the state s is true (or false).*

Proof: Follows from Lemma 1 and Lemma 2. □

The initial values for all the variables in the multithreaded program is assumed to be known by all the threads of the program. Thus, it is assumed that each thread t_i has complete knowledge of the initial values of remote expressions for all processes. These values are then used to initialize the entries $KV_i[j].val$ in the KNOWLEDGEVECTOR of t_i for all j.

3. Examples

To illustrate the expressiveness of the logic, we consider a few standard examples from multithreaded computation literature.

In our first example, we show that we can write formulas which if violated imply a datarace in a multithreaded computation. A datarace occurs when two threads access a shared variable simultaneously without any synchronization and at least one of the accesses is write. Datarace in a multithreaded computation can lead to unexpected behavior which are hard to catch using testing due to their dependency on thread-scheduling. For example, suppose two threads are trying to increment a shared variable x simultaneously by executing the statement $x{+}{+}$ without any synchronization. If the initial value of x is 0 then at the end of the execution the value of x can be 1 or 2 which is erroneous.

Programs containing datarace have been found very difficult to debug as they can exhibit different behaviors under the same set of inputs. It has been recognized that tools capable of detecting dataraces automatically in programs at runtime can be very valuable. In past there has been substantial amount of work to develop tools and techniques to detect dataraces at runtime, such as race detection based tool on "happens-before" relation over locks [8] and Eraser tool [23] based on *locksets*. We next show that we can *precisely* detect dataraces in a way somewhat similar to [8] using our decentralized monitoring approach. We have a clear advantage over the former approach due to our decentralized approach where we divide the detection workload over all the threads. Moreover, we can take a necessary recovery action whenever we find a datarace.

We conservatively say that two accesses of a shared variable x, of which at least one is a write, by two threads are in datarace if we can permute the two accesses without violating the multithreaded computation and make them consecutive. If the two accesses are not consecutive in a given multithreaded execution and we do not know the "happens-before" relation between the different events in the multithreaded execution, then we cannot detect a datarace through simple testing. However, using our monitoring approach we can easily detect such violations if we monitor the following property for every shared variable x and for every pair of threads t_i and t_j in the program

$$\begin{aligned}@_i((\ & \mathrm{read}(x) \wedge \Diamond\, \mathrm{write}(x) \rightarrow \neg @_j(\mathrm{write}(x))) \\ & \wedge(\mathrm{write}(x) \rightarrow \neg @_j(\mathrm{read}(x) \wedge \Diamond\, \mathrm{write}(x))) \\ & \wedge(\mathrm{write}(x) \rightarrow \neg @_j(\mathrm{write}(x))))\end{aligned}$$

where the boolean variable read(x) (or write(x)), local to a thread, is set to *true* whenever the thread reads (or writes) x and set to *false* otherwise. The first conjunct in the formula states the absence of read-write datarace. A read-write datarace happens if a thread reads x and in the past it has written that x and it knows that some other thread has written x in its latest predicted state. Similarly, the second and the third conjunct states the absence of write-read and write-write dataraces. Using our implementation (discussed in Section 4), we were able to precisely detect dataraces in several programs.

The second example considers checking atomicity in a multithreaded Java program. In [11] it has been shown that even if a program is free of dataraces it can have errors arising due to unexpected interaction between threads. Such kind of errors can be due to violation of atomicity requirements. In a given multithreaded computation we say that a block of code is atomic if every interleaving of the execution of the block by a thread with the execution of other threads has the same overall effect as if the block is executed serially by the former thread without interleaving with other threads. This means, we can conservatively say that there is a violation of atomicity in the execution of a block of code, if there is an event from an another thread, interleaved with the execution of the block of code, and the event cannot be permuted with the other events associated with the execution of the block without violating the multithreaded computation. For example, consider the following code executed by two threads t_1 and t_2:

```
int calcBalance(){
  balance1 =                  void transfer(int amount){
checking.balance;               checking.balance = checking.balance
  balance2 =                  - amount;
saving.balance;                 saving.balance = saving.balance +
  return balance1 +           amount;
balance2;                     }
}
```

Thread t_1 executes the method calcBalance and thread t_2 executes the method transfer. We want the execution of both the methods to be atomic. If we have the following execution

t_1: `balance1 = checking.balance;`

t_2: `checking.balance = checking.balance - amount;`

t_1: `balance2 = saving.balance;`

t_2: `saving.balance = saving.balance + amount;`

then there is no violation of atomicity. However, if we have the following execution

t_2: `checking.balance = checking.balance - amount;`

t_1: `balance1 = checking.balance;`

t_1: `balance2 = saving.balance;`

t_2: `saving.balance = saving.balance + amount;`

then there is a violation of atomicity.

One can express simple atomicity requirement elegantly in MtTL as follows:

$$@_i(\mathit{atomic} \to \neg \bigvee_{j \neq i} @_j @_i(\mathit{atomic}))$$

where *atomic* is a boolean variable, local to thread t_i, which set to *true* whenever the thread t_i enters an atomic block of code and set to *false* whenever the thread exits out of the atomic block. The formula states that if thread t_i is in an atomic block then it should not be the case that any other thread knows that thread t_i is in the atomic block. If no other thread knows that thread t_i is in atomic block then the events of that thread interleaved with the atomic block are causally independent of the events in the atomic block. Therefore, there is no violation of atomicity. In the above example, it is easy to see that for the first execution the formula is not violated. However, it is violated in the second execution.

If a thread can execute various atomic blocks several times, then the above formula for atomicity is not sufficient. We need to distinguish various invocations of atomic blocks from each other. We do it by maintaining a counter with every invocation of atomic blocks. The atomicity requirement becomes

$$@_i((\mathit{atomic} > 0) \to \neg \bigvee_{j \neq i} (\mathit{atomic} = @_j @_i \mathit{atomic}))$$

where *atomic* is a local thread variable of type `int`, initialized to 0. Whenever, thread t_i enters an atomic block it sets the variable *atomic* to one plus its last maximum value and sets it to 0 when it comes out of the atomic block. Note that the above formula uses s remote-expression.

Thus using our monitoring approach we can check atomicity property of Java multithreaded programs whose detection at runtime is otherwise considered to be difficult using simple testing.

4. Implementation

We have implemented the above monitoring algorithm in a tool, called DAME [2]. The tool is implemented in Java and can be used to monitor multithreaded Java programs. The tool has two components: an instrumentation tool, and a monitoring library to maintain KnowledgeVector datastructures and perform decentralized monitoring. We next describe the two components.

The instrumentation tool takes a set of Java class files as input. It then instruments the class files as follows. It associates KnowledgeVector with every thread and every shared variable in the program. For every write (or read) of any field of a class, identified by `putfield` (or `getfield`) and `pustatic` (or `getstatic`) instructions in the bytecode, the instrumentation tool inserts code to invoke the KnowledgeVector algorithm for the write event (or read event). In Java two blocks locked by the same lock variable cannot be interleaved, that is, if a block is executed before an another block locked by the same variable then the events in the former block "happens-before" all the events in the latter block. To introduce this "happens-before" relation, lock variables are considered as shared variables and the lock and unlock events are considered as writes to the shared variable.

The instrumentation tool uses BCEL [7] library to systematically instrument every Java class file provided as input. The user of the system provides the name of the class files to be instrumented.

The instrumentation tool also reads the specification file and generates the appropriate class file representing the KnowledgeVector data-structure for that specification. Users can insert arbitrary internal events, such as setting *atomic* to *true* whenever a thread enters an implicit atomic block, by calling a library function `internal(var_name,value)` in the source code of the monitored program.

We monitored several multithreaded Java programs for datarace and atomicity violation. We found bugs related to both datarace and atomicity violation. In particular, in an implementation of a simple banking application (as discussed in section 3), we found atomicity violation in several executions. We noticed 3.4 times slow down when the application was executed to perform around 2000 transactions. This overhead is comparable with the existing tools for atomicity violation detection [11]. In general, in all our experiments we noticed a slow down by a factor of 1.8–5 times. We detected mutual exclusion violation and datarace in a buggy implementation of the mutual exclusion protocol. Our initial experiments suggest the applicability and feasibility of our tool. For more details about the experiments the readers are referred to [2].

5. Related Work

Our work builds on two techniques we developed earlier. These were implemented in the tools JMPaX [25, 24] and DiAna [26]. Specifically, the notion of causality we use was defined in JMPaX and the idea of decentralized monitoring of actor [3] programs was first used in DiAna.

Many researchers have proposed knowledge temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [5] and Halpern *et al.* [9] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [18] and develop methods for model checking formulae in this logic. Our communication primitive is inspired by this work, but we allow arbitrary expressions and atomic propositions over expressions.

Another closely related work is that of Penczek [19, 20] which defines a temporal logic of causal knowledge. Knowledge operators are provided to reason about the local history of a process, as well as about the knowledge it acquires from other processes. However, in order to keep the complexity of model checking tractable, Penczek does not allow the nesting of causal knowledge operators. Interestingly, the nesting of causal knowledge operators does not add any complexity to our algorithm.

Leucker investigates linear temporal logic interpreted over restricted labeled partial orders called Mazurkiewicz traces [15]. An overview of distributed linear time temporal logics based on Mazurkiewicz traces is given by Thiagaranjan *et al.* in [27]. Alur *et al.* [4] introduce a temporal logic of causality (TLC) which is interpreted over causal structures corresponding to partial order executions of a distributed system. They use both past and future time operators and give a model checking algorithm.

In recent years, there has been considerable interest in runtime verification [1]. Havelund *et al.* [13] gives algorithms for synthesizing efficient monitors for safety properties. Sen *et al.* [25] develop techniques for runtime safety analysis for multithreaded programs and introduce the tool JMPaX. Some other runtime verification systems include JPaX from NASA Ames [12] and UPENN's Mac [14].

There has been a substantial work in building tools and techniques that can catch concurrency related bugs such as datarace [23, 8, 6], atomicity violation [11], mutual exclusion violation, and deadlock.

6. Discussion

Our results suggests the utility and feasibility of automatically tracking causal relations in monitoring multithreaded computations. The monitoring helps to detect a number of types of bugs that are a frequent in large multithreaded programs. However, the approach has at least two limitations:

1) Because our monitoring is asynchronous, it is limited to what we have termed robust properties. Not all properties of interest fall into this category: in particular, it is not possible to monitor properties where there is no direct connection between two threads, for example, atomicity properties spanning multiple threads, or relations between values of variables affected by different threads. For such properties, it is necessary to pay the price of synchronous centralized predictive monitors (as in our earlier [24, 25]).

2) The monitoring is entirely syntactic. For example, sometimes concurrent accesses may be permissible because the operations performed do not affect the atomicity property. Because we do not take advantage of the semantics of the operations performed, our detection is overly conservative. Our algorithm needs to be combined with program analysis techniques to determine if a particular observation does indeed result in a semantic safety violation.

References

[1] *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science: 2001, 2002, 2003.

[2] DAME: Decentralized Analyzer of Multithreaded Execution. `http://fsl.cs.uiuc.edu/dame/`, 2005.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[4] R. Alur, D. Peled, and W. Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, 1995.

[5] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, 2001.

[7] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.

[8] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[9] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[10] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (WPDD'88)*, pages 183–194. ACM, 1988.

[11] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL'04)*, pages 256–267, 2004.

[12] K. Havelund and G. Roşu. Java PathExplorer – A runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, 2001.

[13] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer-Verlag, 2002.

[14] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[15] M. Leucker. Logics for Mazurkiewicz traces. Technical Report AIB-2002-10, RWTH, April 2002.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[17] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science, 1989.

[18] B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 487–498. Springer-Verlag, 2000.

[19] W. Penczek. A temporal approach to causal knowledge. *Logic Journal of the IGPL*, 8(1):87–99, 2000.

[20] W. Penczek and S. Ambroszkiewicz. Model checking of causal knowledge formulas. In *Workshop on Distributed Systems (WDS'99)*, volume 28 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1999.

[21] R. Ramanujam. Local knowledge assertions in a changing world. In *Theoretical Aspects of Rationality and Knowledge (TARK'96)*, pages 1–14. Morgan Kaufmann, 1996.

[22] G. Roşu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'04) (Satellite workshop of IPDPS'04)*, Santa Fe, New Mexico, USA, April 2004. IEEE digital library. *Invited Lecture*.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[24] K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *11th International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346. ACM, September 2003.

[25] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138, March 2004.

[26] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 418–427. IEEE, May 2004.

[27] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Twelth Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 183–194, 1997.

A. Proof of Lemma 1:

For any event e, let $KV_i(e)$ be the knowledge vector associated with the thread t_i after the event e. Similarly, let $KV_x^a(e)$ and $KV_x^w(e)$ be the access knowledge vector and write knowledge vector, respectively, associated with the variable x after the event e. Let $e' = w_x(e)$ be the latest write event of x such that $e' \preccurlyeq e$. We show that $KV_i(e_i^k)[x]$ contains information about x after the event $w_x(e_i^k)$, that is

- $KV_i(e_i^k)[x].seq$ contains the number of writes to the variable x till the event $w_x(e_i^k)$ (including $w_x(e_i^k)$),
- $KV_i(e_i^k)[x].val$ contains the value of x after the event $w_x(e_i^k)$.

We prove this by induction by considering the following cases.

1. e_i^k **is write of** x**:** Let $KV_j(e_j^l)[x].seq = KV_x^a(e_j^l)[x].seq = KV_x^w(e_j^l)[x].seq = n$, where e_j^l is the latest write event of x such that $e_j^l \prec e_i^k$. If we assume that our hypothesis holds for e_j^l, then n is the number of writes to x till e_j^l (including e_j^l). After the event e_i^k, since KnowledgeVector algorithm sets $KV_i(e_i^k)[x].seq = KV_x^a(e_i^k)[x].seq = KV_x^w(e_i^k)[x].seq = n+1$, $KV_i(e_i^k)[x].seq$ equals the number of writes to x till e_i^k (including e_i^k). Moreover, $KV_i(e_i^k)[x].val$ is updated by the value of x written by e_i^k. Therefore, our hypothesis holds for e_i^k.

2. e_i^k **is read of** x**:** Let $KV_j(e_j^l)[x].seq = KV_x^a(e_j^l)[x].seq = KV_x^w(e_j^l)[x].seq = n$, where $e_j^l = w_x(e_i^k)$. If we assume that our hypothesis holds for e_j^l, then n is the number of writes to x till e_j^l (including e_j^l). After the event e_i^k, since KnowledgeVector algorithm sets $KV_i(e_i^k)[x].seq = KV_x^a(e_i^k)[x].seq = n$, $KV_i(e_i^k)[x].seq$ equals the number of writes to x till e_i^k. Therefore, our hypothesis holds for e_i^k.

3. e_i^k **is write of** $y(\neq x)$**:** Let $E = \{e \mid e \text{ read of } y \text{ and } e \lessdot e_i^k\}$, then $E' = E \cup \{e_i^{k-1}\}$ is the set of all events that "happen-before" ($\lessdot$) e_i^k. Therefore, $w_x(e_i^k)$ is the latest event in the set $\{w_x(e) \mid e \in E'\}$. By the KnowledgeVector algorithm and induction hypothesis $KV_y^a[x]$ just before e_i^k contains information about x after the latest event in the set $\{w_x(e) \mid e \in E\}$. This is because after every event $e \in E$, if the event belongs to thread t_j then $KV_y^a[x]$ is updated with $\max(KV_y^a[x], KV_j[x])$. Therefore, when KnowledgeVector algorithm updates $KV_i(e_i^k)[x]$ with the maximum of $KV_i(e_i^{k-1})[x]$ and $KV_y^a[x]$ just before the event e_i^k, $KV_i(e_i^k)[x]$ contains information about $w_x(e_i^k)$.

4. e_i^k **is read of** $y(\neq x)$**:** Then $\{e_i^{k-1}\} \cup \{w_y(e_i^k)\}$ is the set of all events that "happen-before" ($\lessdot$) e_i^k. If $w_y(e_i^k) = e'$ (say) belongs to thread t_j, then $KV_y^w(e')[x] = KV_j(e')[x]$ by the KnowledgeVector algorithm. Therefore, after the event e_i^k, when $KV_i(e_i^k)[x]$ is updated by $\max(KV_y^w[x], KV_i[x])$ or $\max(KV_y^w(e')[x], KV_i(e_i^{k-1})[x])$, $KV_i(e_i^k)$ contains the information about x which is latest after the events e' and e_i^{k-1}.

$\square$