Efficient Decentralized Monitoring of Safety in Distributed Systems [1]

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

We describe an efficient decentralized algorithm to monitor the execution of a distributed program in order to check for violations of safety properties. The monitoring is based on formulas written in PT-DTL, a variant of past time linear temporal logic that we define. PT-DTL is suitable for expressing temporal properties of distributed systems. Specifically, the formulas of PT-DTL are relative to a particular process and are interpreted over a projection of the trace of global states that represents what that process is *aware of*. A formula relative to one process may refer to the local states of other processes through remote expressions and remote formulas. In order to correctly evaluate remote expressions, we introduce the notion of *knowledge vector* and provide an algorithm which keeps a process aware of other processes' local states, if those states may affect the validity of a monitored PT-DTL formula. Both the logic and the monitoring algorithm are illustrated through a number of examples. Finally, we describe our implementation of the algorithm in a tool called DIANA.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Verification
Additional Key Words and Phrases: Actors, Distributed systems, Decentralized analysis, Runtime monitoring, Happens-before relation, Knowledge vector, Vector clock.

1. INTRODUCTION

Software errors arise from a number of different problems, such as incorrect or incomplete specifications, coding errors, and faults and failures in the hardware, operating system or network. Model checking [E. M. Clarke et al. 1999] is an important technology which is finding increasing use as a means of reducing software errors. Unfortunately, despite impressive recent advances, the size of systems for which model checking is feasible remains rather limited. This weakness is particularly critical in the context of distributed systems: concurrency and asynchrony result in inherent non-determinism that significantly increases the number of states to be analyzed. As a result, most system builders continue to use testing as the major means to identify bugs in their implementations.

There are, however, two problems with software testing. First, testing is generally done in an *ad hoc* manner: the software developer must hand translate the requirements into specific dynamic checks on the program state. Second, test coverage is often rather limited, covering only some execution paths. To mitigate the first problem, software often includes dynamic checks on a system's state in order to identify problems at run-time. Recently, there has been some interest in run-time verification and monitoring [Havelund and Roşu

[1] Part of the work reported in this paper was presented at the 26^{th} International Conference on Software Engineering (ICSE'04) [Sen et al. 2004a].

2004; Sokolsky and Viswanathan 2003] techniques, which provide a little more rigor in testing by automatically synthesizing monitors from formal specifications. These monitors may then be deployed off-line for debugging or on-line for dynamically checking that safety properties are not being violated during system execution.

Unfortunately, testing or runtime monitoring of distributed systems involves considerable overhead: for every event (sending of a message, receiving of a message, or a local state update), each process sends a message about the event to a central monitor. The central monitor constructs and analyzes a computation lattice [Babaoğlu and Marzullo 1993] of the global states out of the collected events. Passing messages to a central monitor at every event and constructing a global computation lattice, which can be exponential in size in the number of events, leads to severe communication and computation overhead. In the present work, we argue that distributed systems may be effectively monitored at runtime against formally specified safety requirements by *distributing the task of monitoring* among the processes involved in the distributed computation. By effective monitoring, we mean not only linear efficiency, but also decentralized monitoring where few or no additional messages need to be passed for monitoring purposes.

We introduce an epistemic temporal logic for distributed knowledge and illustrate the expressiveness of this logic by means of several examples. We then show how efficient distributed monitors can automatically be synthesized from requirements specified in this logic. Finally, we present a software system implementing the proposed techniques, as a development and monitoring framework for distributed systems applications, called DIANA. To use DIANA, a user must provide an application together with the formal safety properties that he or she wants monitored. DIANA automatically synthesizes code for monitoring the specified requirements and weaves appropriate instrumentation code into the given application. As soon as a safety violation is revealed by any of the local monitors at runtime, user-provided recovery code can be executed; that code is intended to bring the system back to a safe state by, for example, rebooting it or releasing its resources.

The work presented in this paper was stimulated by the observation that, in distributed systems, it is generally impractical to monitor requirements expressed in classical temporal logics. For example, consider a system of mobile nodes in which one mobile node may request a certain value from another node. On receiving the request, the second node computes the value and returns it. An important requirement in such a system is that no node receives a reply from a node to which it has not previously issued a request. It is easy to see that Linear Temporal Logic (LTL) would not be a practical specification language for any reasonably sized collection of nodes. To use LTL, we would need to collect consistent snapshots of the global system; a central monitor would then check the snapshots for possible violations of the property by considering all possible interleavings of events that are allowed by the distributed computation. In a system of thousands of nodes, collecting such a global snapshots would be prohibitive. Moreover, the number of possible interleavings to be considered would be large even if powerful techniques such as partial order reduction are used. To address the above difficulty, we define *past-time distributed temporal logic* (PT-DTL). Using PT-DTL, one can check/monitor a global property such as the one above by checking/monitoring a local property at each node.

The work presented in this paper brings at least three major contributions. First, we define a simple but expressive logic to specify safety properties in distributed systems. Second, we provide an algorithm to synthesize decentralized monitors for safety properties

that are expressed in the logic. Finally, we describe the implementation of a tool (DIANA) that is based on this technique.

The paper is organized as follows. Section 2 gives some motivating examples and informally introduces PT-DTL. Section 3 and Section 4 give the preliminaries. Section 5 formally introduces PT-DTL. In Section 6, we describe the algorithm that underlies our implementation. Section 7 briefly describes the implementation along with initial experimentation.

2. MOTIVATING EXAMPLES

Let us assume an environment in which a node a may send a message to a node b requesting a certain value. The node b, on receiving the request, computes the value and sends it back to a. There can be many such nodes, any pair can be involved in such a transaction, but suppose that a crucial property to enforce is that no node receives a reply from another node to which it had not issued a request earlier. One can check this global property by having one local monitor on each node, which monitors a single property. For example, node a monitors "if a has received a value then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past". This is precisely and concisely expressed by the PT-DTL formula:

$$@_a(\texttt{receivedValue} \rightarrow @_b(\Diamond(\texttt{computedValue} \wedge @_a(\Diamond\texttt{requestedValue}))))$$

Note that we read $@$ as "at", $@_b F$ is the value of F in the most recent local state of b that the current process is aware of, and $\Diamond$ denotes the formula was true sometime in the past. Like in [Sen et al. 2004b], $@$ is allowed to take any set of processes as a subscript together with a universal or an existential quantifier; therefore, $@_b$ becomes "syntactic sugar" for $@_{\forall\{b\}}$ (or for $@_{\exists\{b\}}$). Monitoring the above formula involves sending no additional messages – it involves inserting only a few bits of information which are piggybacked on the messages that are already being passed in the computation. This efficiency provides a substantial improvement over what is required to monitor formulas written in classical LTL.

Moreover, we introduce *remote expressions* in PT-DTL to refer to numerical values depending on the state of a remote process. For example, a process a may monitor the property: "if my alarm has been set then it must be the case that the difference between my temperature and the temperature at process b exceeded the allowed value":

$$@_a(\texttt{alarm} \rightarrow \Diamond((\texttt{myTemp} - @_b\texttt{otherTemp}) > \texttt{allowed}))$$

$@_b\texttt{otherTemp}$ is a remote expression that is subtracted from the local value of myTemp.

An example of a safety property that may be useful in the context of an airplane software is: "if my airplane is landing then the runway allocated by the airport matches the one that I am planning to use". This property may be expressed in PT-DTL as follows:

$$@_{\texttt{airplane}}(\texttt{landing} \rightarrow (\texttt{runway} = (@_{\texttt{airport}}\texttt{allocRunway})))$$

Another example considers monitoring a correctness requirement in a leader-election algorithm. The key requirement for leader election is that there is at-most one leader. If there are 3 processes, say a, b, c, and state is a variable in each process that can have values leader, loser, candidate, sleep, then we can write the property at every process

as: "if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader". We can formalize this requirement as the following PT-DTL formula at process a:

$$@_a(\texttt{leaderElected} \rightarrow (\texttt{state} = \texttt{leader} \rightarrow (@_b(\texttt{state} \neq \texttt{leader}) \wedge @_c(\texttt{state} \neq \texttt{leader}))))$$

We can write similar formulas with respect to b and c. Given an implementation of the leader election problem, one can monitor each formula locally, at every process. If violated then clearly the leader election implementation is incorrect.

Note that the above formula assumes that the name of every process involved in leader election is known a priori. Moreover, the size of the formula depends on the number of processes. In a distributed system involving a large number of processes, writing such a large formula may be impractical. The problem becomes even more important in an *evolving* distributed system (new processes are created and destroyed dynamically) where one may not know the name of processes beforehand. To alleviate this difficulty, as already mentioned, we use a set of indices instead of a single index in the operator $@$. The set of indices denoting a set of processes can be represented compactly by a predicate on indices. For example, in the above formula, instead of referring to each process by its name, we can refer to the set of all remote processes by the predicate $j \neq i$ and use this set as a subscript to the operator $@$:

$$@_i(\texttt{leaderElected} \rightarrow (\texttt{state} = \texttt{leader} \rightarrow @_{\forall\{j|j\neq i\}}(\texttt{state} \neq \texttt{leader})))$$

3. DISTRIBUTED SYSTEMS

A distributed system is a collection of n processes $(p_1, \ldots, p_n)$, each with its own local state. The local state of a process is given by the values bound to its variables. Note that there are no global or shared variables. Processes communicate with each other using asynchronous messages whose order of arrival is indeterminate. The computation of each process is abstractly modelled by a set of *events*, and a distributed computation is specified by a partial order $\prec$ on the events. There are three types of events:

(1) *internal* events change the local state of a process;
(2) *send* events occur when a process sends a message to another process; and
(3) *receive* events occur when a message is received by a process.

Let E_i denote the set of events of process p_i and let E denote $\bigcup_i E_i$. Now, $\lessdot \subseteq E \times E$ is defined as follows:

(1) $e \lessdot e'$ if e and e' are events of the same process and e happens immediately before e',
(2) $e \lessdot e'$ if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. This partial order captures the *causality* relation between events. The structure described by $\mathcal{C} = (E, \prec)$ is called a *distributed computation* and we assume an arbitrary but given distributed computation $\mathcal{C}$. Further, $\preccurlyeq$ is the reflexive and transitive closure of $\lessdot$.

As an illustration, in Fig. 1, $e_{11} \prec e_{23}$, $e_{12} \prec e_{23}$, and $e_{11} \lessdot e_{23}$. However, $e_{12} \not\lessdot e_{23}$.

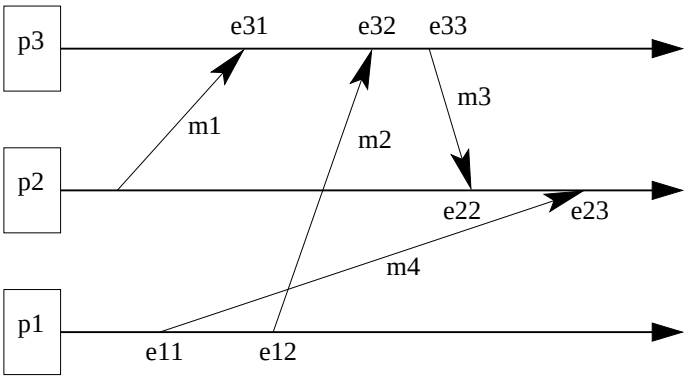

Fig. 1. Sample Distributed Computation

For $e \in E$, we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \preccurlyeq e\}$, that is, $\downarrow e$ is the set of events that causally precede e. For $e \in E_i$, we can think of $\downarrow e$ as the local state of p_i when the event e has just occurred. This state contains the history of events of all processes that causally precede e.

We extend the definition of $\lessdot$, $\prec$ and $\preccurlyeq$ to local 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 local states of a process p_i by $LS_i \stackrel{\text{def}}{=} \{\downarrow e \mid e \in E_i\}$ and let $LS \stackrel{\text{def}}{=} \bigcup_i LS_i$. We use the symbols $s_i, s_i', s_i'', \ldots$ to represent the local states of process p_i. We also assume that the local state s_i of each process p_i associates values to some local variables V_i, and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process p_i.

We use the notation $@_j(s_i)$ to refer to the latest state of process p_j that the process p_i knows while in state s_i. Formally, if $@_j(s_i) = s_j$ then $s_j \in LS_j$ and $s_j \preccurlyeq s_i$ and for all $s_j' \in LS_j$ if $s_j' \preccurlyeq s_i$ then $s_j' \preccurlyeq s_j$. For example, in Figure 1 $@_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$ then $@_j(s_i) = s_i$.

4. PAST TIME LINEAR TEMPORAL LOGIC (PT-LTL)

Past-time Linear Temporal Logic (PT-LTL) [Manna and Pnueli 1992; 1995] has been used in [Havelund and Roşu 2002; Kim et al. 2001; Sen et al. 2003] to express, monitor and predict violations of safety properties of software systems. The syntax of PT-LTL is:

$$\begin{array}{rll} F ::= & \text{true} \mid \text{false} \mid a \in A \mid \neg F \mid F \; op \; F & \text{propositional} \\ & \mid \odot F \mid \Diamond F \mid \Box F \mid F \; \mathcal{S} \; F & \text{temporal} \end{array}$$

where op are standard binary operators, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. $\odot F$ should be read as "previously", $\Diamond F$ as "eventually in the past", $\Box F$ as "always in the past", $F_1 \mathcal{S} F_2$ as "F_1 since F_2". The logic is interpreted on a finite sequence of states or a *run*. If $\rho = s_1 s_2 \ldots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table I. For example, the formula $\Box((\texttt{action} \wedge \odot\neg\texttt{action}) \rightarrow (\neg\texttt{stop} \; \mathcal{S} \; \texttt{start}))$ states that whenever `action` starts to be true, it is the case that `start` was true at some point in the past and since then `stop` was never true: in other words, the action is taken only while the system is active. Notice that the semantics of "previously" is given as if the trace is unbounded in the past and stationary in the first event. In runtime monitoring, we start the process of monitoring from

$\rho \models \text{true}$	for all ρ,
$\rho \not\models \text{false}$	for all ρ,
$\rho \models a$	iff a holds in the state s_n,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \ op \ F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \Box F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \ \mathcal{S} \ F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $j < i \leq n$,

Table I. Semantics of PT-LTL

the point that the first event is generated and we continue monitoring for as long as events are generated.

Although PT-LTL is interpreted over a linear execution trace, in distributed systems a computation is a partial order which may have several possible linearizations. Therefore, monitoring a distributed computation requires monitoring all possible linear traces that may be obtained from a partial order. Unfortunately, the number of linearizations of a partial order may be exponential in the length of the computation and thus monitoring a PT-LTL formula may become easily intractable. A major contribution of this paper is to extend PT-LTL so that we can reason about a distributed property using only local monitoring. We describe this extension next.

5. PAST TIME DISTRIBUTED TEMPORAL LOGIC

Although PT-LTL works well for a single process, once we have more processes interacting with each other we need to reason about the state of remote processes. Since practical distributed systems are usually asynchronous and the absolute global state of the system is *not* available to processes, the best thing that each process can do it to reason about the global state that it is *is aware of*.

We define Past-Time Distributed Temporal Logic (PT-DTL) by extending PT-LTL to express safety properties of distributed message passing systems. Specifically, we add a pair of *epistemic operators* as in [Ramanujam 1996], written $@$, whose role is to evaluate an expression or a formula in the *last known state* of a set of remote processes. We call such an expression or a formula *remote*. A remote expression or formula may contain nested epistemic operators and refer to variables that are local to a remote process. By using remote expressions, in addition to remote formulas, a larger class of desirable properties of distributed systems may be specified without sacrificing the efficiency of monitoring.

For example, consider the simple local property at a process p_i that if α is true in the current local state of p_i then β must be true at the latest state of process p_j of which p_i is aware of. This property will be written formally in PT-DTL as $@_i(\alpha \rightarrow @_j\beta)$. However, referring to remote formulas only is *not* sufficient to express a broad range of useful global properties such as "at process p_i, the value of x in the current state is greater than the value of y at process p_j in the latest causally preceding state." The reason we introduce the novel epistemic operators on expressions is that it is crucial to be able to also refer to *values* of expressions in remote local states. For example, the property above can be formally

specified as the PT-DTL formula $@_i(x > @_j y)$ at process p_i where $@_j y$ is the value of y at process p_j that p_i is aware of.

The intuition underlying PT-DTL is that each process is associated with local temporal formulas which may refer to the global state of the distributed system. These formulas are required to be valid at the respective processes during a distributed computation. A distributed computation satisfies the specification when all the local formulas are shown to satisfy the computation.

5.1 Syntax

From now on, we will only consider PT-DTL formulas associated to, or *local to*, individual processes. We call such formulas associated to process p_i *i-formulas* and let $F_i, F_i', \ldots$ denote them. Further, we introduce *i-expressions* as expressions that are local to a process p_i and let $\xi_i, \xi_i', \ldots$ denote them. Informally, an i-expression is an expression over the global state of the system that process p_i is currently aware of. Local predicates on i-expressions form the atomic propositions on which the temporal i-formulas are built.

We add the *epistemic operators* $@_{\forall J} F_j$ and $@_{\exists J} F_j$ which is true if at all (or some, respectively) processes j in the set J, F_j holds. Similarly, we add the epistemic operator $@_J \xi_j$ which returns the set of j-expressions ξ_j for all processes j in the set J. The sets J can be expressed compactly using predicates over j. For example, J can be the sets $\{j \mid j \neq a\}$ or $\{j \mid \texttt{client}(j)\}$. The following gives the formal syntax of PT-DTL, where i and j are names of any process (not necessarily distinct):

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

A top-level PT-DTL formula F is always of the form $@_i F_i$ implying that it is always specified local to a process. The infix operator *op* may be a binary propositional operator such as $\wedge, \vee, \rightarrow$ or $\equiv$. The term $\vec{\xi_i}$ stands for a tuple of expressions on process p_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 may be $<, \leq, >, \geq, =$ and f may be $+, -, /, *$. Variables v_i belongs to the set V_i which contains all the local state variables of process p_i. Constants such as $0,\ 1,\ 3.14$ are represented by $c, c', c_1, \ldots$.

The expression $@_j \xi_j$ is syntactic sugar for $elem(@_{\{j\}} \xi_j)$, where the function $elem$ takes a set containing a single expression and returns that expression. Similarly, $@_j F_j$ is syntactic sugar for either $@_{\forall\{j\}} F_j$ or $@_{\exists\{j\}} F_j$ (they are equivalent).

5.2 Semantics

The semantics of PT-DTL is a natural extension of PT-LTL with the intuitive behavior for the epistemic operators. The atomic propositions of PT-LTL are replaced by predicates over tuples of expressions. Table II formally gives the semantics of each operator of PT-DTL. $(\mathcal{C}, s_i)[\![@_J \xi_j]\!]$ is the set of values of the expression ξ_j in the state $s_j = @_j(s_i)$ which is the latest state of process p_j for each $j \in J$ of which process p_i is aware of. We assume

that expressions are properly typed. Typically, these types could be: `integer, real, strings`. We assume that $s_i, s_i', s_i'', \ldots \in LS_i$ and $s_j, s_j', s_j'', \ldots \in LS_j$. Notice that, as in PT-LTL, 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*.

$\mathcal{C}, s_i \models \text{true}$	for all s_i
$\mathcal{C}, s_i \not\models \text{false}$	for all s_i
$\mathcal{C}, s_i \models P(\xi_i, \ldots, \xi_i')$	iff $P((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!]) = \text{true}$
$\mathcal{C}, s_i \models \neg F_i$	iff $\mathcal{C}, s_i \not\models F_i$
$\mathcal{C}, s_i \models F_i \; op \; F_i'$	iff $\mathcal{C}, s_i \models F_i$ op $\mathcal{C}, s_i \models F_i'$
$\mathcal{C}, s_i \models \odot F_i$	iff if $\exists s_i' \,.\, s_i' \lessdot s_i$ then $\mathcal{C}, s_i' \models F_i$ else $\mathcal{C}, s_i \models F_i$
$\mathcal{C}, s_i \models \Diamond F_i$	iff $\exists s_i' \,.\, s_i' \preccurlyeq s_i$ and $\mathcal{C}, s_i' \models F_i$
$\mathcal{C}, s_i \models \Box F_i$	iff $\mathcal{C}, s_i \models F_i$ for all $s_i' \preccurlyeq s_i$
$\mathcal{C}, s_i \models F_i \; \mathcal{S} \; F_i'$	if $\exists s_i' \,.\, s_i' \preccurlyeq s_i$ and $\mathcal{C}, s_i' \models F_i'$ and $\forall s_i'' \,.\, s_i' \prec s_i'' \preccurlyeq s_i$ implies $\mathcal{C}, s_i'' \models F_i$
$\mathcal{C}, s_i \models @_{\forall J} F_j$	iff $\forall j \,.\, (j \in J) \rightarrow \mathcal{C}, s_j \models F_j$ where $s_j = @_j(s_i)$
$\mathcal{C}, s_i \models @_{\exists J} F_j$	iff $\exists j \,.\, (j \in J) \wedge \mathcal{C}, s_j \models F_j$ where $s_j = @_j(s_i)$

$(\mathcal{C}, s_i)[\![v_i]\!]$	$= s_i(v_i)$, that is, the value of v_i in s_i
$(\mathcal{C}, s_i)[\![c_i]\!]$	$= c_i$
$(\mathcal{C}, s_i)[\![f(\xi_i, \ldots, \xi_i')]\!]$	$= f((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!])$
$(\mathcal{C}, s_i)[\![@_J \xi_j]\!]$	$= \{(\mathcal{C}, s_j)[\![\xi_j]\!] \mid \; s_j = @_j(s_i) \wedge j \in J\}$

Table II. Semantics of PT-DTL

5.3 Examples

To illustrate PT-DTL, we consider a few relatively simple examples. The first example is concerned with *majority vote*. The desired property, "if the resolution is accepted then more than half of the processes say yes", can be stated as:

$$@_i(\texttt{accepted} \rightarrow \texttt{sum}(@_{\{j \mid j \text{ is any process}\}}(\texttt{vote}))) > n/2)$$

where a process stores 1 in a local variable `vote` if it is in favor of the resolution, and 0 otherwise; `sum` is a function that takes as argument a set of values and returns their sum.

A second example is a safety property that a server must satisfy in case it reboots itself: "the server accepts the command to reboot only after knowing that each client is inactive and aware of the warning about pending reboot." The property is expressed as the *server*-local formula below which contains nested epistemic operators:

$$\texttt{rebootAccepted} \rightarrow \bigwedge\nolimits_{\texttt{client}}(@_{\texttt{client}}(\texttt{inactive} \wedge @_{\texttt{server}}\texttt{rebootWarning}))$$

6. MONITORING ALGORITHM FOR PT-DTL

We next describe a technique to automatically synthesize efficient distributed monitors for safety properties of distributed systems expressed in PT-DTL. We assume that one or more processes are associated with PT-DTL formulas that must be satisfied by the distributed computation. The synthesized monitor is *distributed*, in the sense that it consists of separate, *local monitors* running on each process. A local monitor may attach additional information to an outgoing message from the corresponding process. This information can subsequently be extracted by the monitor on the receiving side without changing the underlying semantics of the distributed program. The key guiding principles in the design of this technique are as follows:

—A local monitor should be fast, so that monitoring can be done online;

—A local monitor should have little memory overhead, in particular, it should *not* need to store the entire history of events on a process; and

—The number of messages that need to be sent between processes for the purpose of monitoring should be minimal.

In this section, when we refer to a remote expression or formula we mean one which occurs in any of the monitored PT-DTL formulas.

6.1 Knowledge Vectors

Consider the problem of evaluating a remote j-expression $@_j\xi_j$ at process p_i. A naive solution is that process p_j simply piggybacks the value of ξ_j with every message that it sends out. The recipient process p_i can extract this value and use it as the value of $@_j\xi_j$. However, this approach is problematic: recall that messages from p_j could reach p_i in an arbitrary order: because the arrival order of two messages, even from the same sender, is indeterminate, more recent values may be overwritten by older ones. To keep track of the causal history, or in other words the most recent knowledge, we add an event number corresponding to the local history sequence at p_j at the time expressions were sent out in messages. Stale information in a reordered message sequence is then simply discarded.

Causal ordering can be effectively accomplished by using an array called KNOWLEDGEVECTOR with an entry for any process p_i participating in the distributed computation. Knowledge vectors are motivated and inspired by vector clocks [Fidge 1988; Mattern 1989]. Let $KV[j]$ denote the entry for process p_j on a vector KV. $KV[j]$ contains the following fields:

—The sequence number of the last event seen at p_j, denoted by $KV[j].seq$;

—A set of values $KV[j].values$ storing the values j-expressions and j-formulas.

Each process p_i keeps a local KNOWLEDGEVECTOR denoted by KV_i. The monitor of process p_i attaches a copy of KV_i with every outgoing message m. We denote the copy by KV_m. The algorithm for the update of KNOWLEDGEVECTOR KV_i at process p_i is as follows:

(1) **[internal]:** update $KV_i[i]$. Evaluate $eval(\xi_i, s_i)$ and $eval(F_i, s_i)$ (see Subsection 6.2) for each $@_I\xi_i$ and $@_{\forall I}F_i$ (or $@_{\exists I}F_i$), respectively, and store them in the set $KV_i[i].values$;

(2) **[send** m**]:** $KV_i[i].seq \leftarrow KV_i[i].seq + 1$. Send KV_i with m as KV_m;

$$
\begin{array}{lcl}
\mathcal{C}, s_i \models \Diamond F_i & = & \mathcal{C}, s_i \models F_i \text{ or } (\exists s'_i \,.\, s'_i \lessdot s_i \text{ and } \mathcal{C}, s'_i \models \Diamond F_i) \\
\mathcal{C}, s_i \models \Box F_i & = & \mathcal{C}, s_i \models F_i \text{ and } (\exists s'_i \,.\, s'_i \lessdot s_i \text{ implies } \mathcal{C}, s'_i \models \Box F_i) \\
\mathcal{C}, s_i \models F_i \mathcal{S} F'_i & = & \mathcal{C}, s_i \models F'_i \text{ or} \\
 & & (\mathcal{C}, s_i \models F_i \text{ and } \exists s'_i \,.\, s'_i \lessdot s_i \text{ and } \mathcal{C}, s'_i \models F_i \mathcal{S} F'_i)
\end{array}
$$

Table III. Recursive Semantics of PT-DTL

(3) **[receive** m**]:** for all j, if $KV_m[j].seq > KV_i[j].seq$ then $KV_i[j] \leftarrow KV_m[j]$, that is,
$KV_i[j].seq \leftarrow KV_m[j].seq$,
$KV_i[j].values \leftarrow KV_m[j].values$.
Evaluate $eval(\xi_i, s_i)$ and $eval(F_i, s_i)$ for each $@_I \xi_i$ and $@_{\forall I} F_i$ (or $@_{\exists I} F_i$), respectively, and store them in the set $KV_i[i].values$.

We call this the KNOWLEDGEVECTOR algorithm. Informally, $KV_i[j].values$ contains the latest values that p_i has for j-expressions or j-formulas. Therefore, for the value of a remote expression or formula of the form $@_J \xi_j$ or $@_{\forall J} F_j$ (or $@_{\exists J} F_j$), process p_i can just use the entry corresponding to ξ_j or F_j in the set $KV_i[j].values$. Note that the sequence number needs to be incremented only when sending messages. The correctness of the algorithm can be stated as the following proposition.

PROPOSITION 6.1. *For any process p_i and any j, the entry for ξ_j or F_j in $KV_i[j].values$ contains the value of $@_j \xi_j$ or $@_j F_j$, respectively.*

The initial values for all the variables in a distributed program may be found either by a static analysis of the program or by a distributed broadcast at the beginning of the computation. Thus, it is assumed that each process p_i has the complete knowledge of the initial values of remote expressions for all processes. These values are used to initialize the entries $KV_i[j].values$ in the KNOWLEDGEVECTOR of p_i for all j.

6.2 Monitoring a Local PT-DTL Formula

The monitoring algorithm for a PT-DTL formula is similar in spirit to that for an ordinary PT-LTL formula described in [Sen et al. 2003]. The key difference is that we allow remote expressions and remote formulas whose values and validity, respectively, need to be transferred from the remote process to the current process. Once the KNOWLEDGEVECTOR is properly updated, the local monitor can compute the boolean value of the formula to be monitored, by recursively evaluating the boolean value of each of its subformulas in the current state. To do so, it may also use the boolean values of subformulas evaluated in the previous state and the values of remote expressions and remote formulas.

The function *eval* is defined next. *eval* takes advantage of the recursive nature of the temporal operators (see Table III) to calculate the boolean value of a formula in the current state in terms of (a) its boolean value in the previous state and (b) the boolean value of its subformulas in the current state. The function *op(F_i)* returns the operator of the formula F_i, *binary(op(F_i))* returns *true* if *op(F_i)* is binary, *unary(op(F_i))* returns *true* if *op(F_i)* is unary, *left(F_i)* returns the left subformula of F_i, *right(F_i)* returns the right subformula of F_i when *op(F_i)* is binary, and *subformula(F_i)* returns the subformula of F_i otherwise. The variable index represents the index of a subformula.

array now; **array** pre; **int** $index$;

boolean $eval(Formula\ F_i, \textbf{State}\ s_i)\{$
  if $binary(op(F_i))$ **then**$\{$
    $lval \leftarrow eval(left(F_i), s_i)$;
    $rval \leftarrow eval(right(F_i), s_i)$;$\}$
  else if $unary(op(F_i))$ **then**
    $val \leftarrow eval(subformula(F_i), s_i)$;
  $index \leftarrow 0$;
  case$(op(F_i))$ **of**$\{$
    $true$: **return** $true$; $\ false$: **return** $false$;
    $P(\vec{\xi_i})$: **return** $P(eval(\xi_i, s_i), \ldots, eval(\xi_i', s_i)))$;
    op : **return** $rval\ op\ lval$; $\neg$: **return not** val;
    $\mathcal{S}$: $now[index] \leftarrow (pre[index]$ **and** $lval)$ **or** $rval$;
      return $now[index{+}{+}]$;
    $\boxdot$: $now[index] \leftarrow pre[index]$ **and** val;
      return $now[index{+}{+}]$;
    $\diamondsuit$: $now[index] \leftarrow pre[index]$ **or** val;
      return $now[index{+}{+}]$;
    $\odot$: $now[index] \leftarrow val$; **return** $pre[index{+}{+}]$;
    $@_{\forall J} F_j$: **return** $\bigwedge_{j \in J} F_j$ where value of F_j is looked up from $KV_i[j].values$;
    $@_{\exists J} F_j$: **return** $\bigvee_{j \in J} F_j$ where value of F_j is looked up from $KV_i[j].values$;
  $\}$
$\}$

where the global array *pre* contains the boolean values of all subformulas in the previous state that will be required in the current state, while the global array now, after the evaluation of *eval*, will contain the boolean values of all subformulas in the current state that may be required in the next state. Note that the *now* array's value is set in the function *eval*. The function *eval* on expressions is defined next.

value $eval(\textbf{Expression}\ \xi_i, \textbf{State}\ s_i)\{$
  case(ξ_i) **of**$\{$
    v_i: **return** $s_i(v_i)$; c_i: **return** c_i;
    $f(\xi_i^1, \ldots, \xi_i^k)$: **return** $f(eval(\xi_i^1, s_i), \ldots, eval(\xi_i^k, s_i))$;
    $@_J \xi_j'$: **return** $\{\xi_j' \mid j \in J\}$ where value of ξ_j' is looked up from $KV_i[j].values$;
  $\}$
$\}$

Note that the function *eval* cannot be used to evaluate the boolean value of a formula at the first event, as the recursion handles the case $n = 1$ in a different way. We define the function *init* to handle this special case as implied by the semantics of PT-DTL in Tables II and III on one event traces.

boolean $init(Formula\ F_i, \textbf{State}\ s_i)\{$
  if $binary(op(F_i))$ **then**$\{$
    $lval \leftarrow init(left(F_i, s_i))$;
    $rval \leftarrow init(right(F_i, s_i))$;$\}$
  else if $unary(op(F_i))$ **then**
    $val \leftarrow init(subformula(F_i, s_i))$;

```
  index ← 0;
  case(op(F_i)) of{
    true : return true;  false : return false;
    P(ξ⃗_i) : return P(eval(ξ_i, s_i), ..., eval(ξ'_i, s_i)));
    op :  return rval op lval; ¬ :  return not val;
    S :  now[index] ← rval;  return now[index++];
    □, ◇, ⊙ :  now[index] ← val;  return now[index++];
  }
}
```

For a PT-DTL formula $@_i F_i$, we call p_i *the owner* of that formula. At the owner process, we evaluate F_i using the $eval$ function after every internal or receive event, and then assign *now* to *pre*. This is done after the KNOWLEDGEVECTOR is correspondingly updated after the event. If the evaluation of F_i at process p_i is false then we report a warning that the formula $@_i F_i$ has been violated. The time and space complexity of this algorithm at every event is $\Theta(mn)$, where m is the size of the original local formula and n is the number of processes involved in the distributed computation.

The evaluation of a formula $@_{\forall J} F_j$ (or $@_{\exists J} F_j$) or an expression $@_J \xi_j$ requires the computation of the set J at runtime. If the elements of the set depend on the dynamic behavior of the program, then every knowledge vector needs to maintain the value of the formula F_j or of the expression $@_J \xi_j$ for every process j involved in the computation. This implies that the size of the knowledge vector is linear in the number of the processes in the system. However, if the set J can be determined syntactically before the program starts, a knowledge vector needs only to maintain the values of the F_j of $@_J \xi_j$ for the processes in the set J. In particular, if a formula is of the form $@_a F_a$, where a is some unique process in the system, a knowledge vector only needs to maintain the entry F_a. This implies that the size of a knowledge vector can be independent o the number of the processes in the system if all the subformulas having @ operator at the top are of the form $@_j F_j$ or $@_J \xi_j$. This particular case was the one presented in [Sen et al. 2004a].

6.3 Example

To illustrate the monitoring algorithm, we consider a simple example where all the subformulas with the epistemic operator @ at the top are of the form $@_j F_j$ or $@_J \xi_j$. Consider three processes, p_1, p_2 and p_3. p_1 has a local variable x whose initial value is 5, p_2 has a local variable y with initial value 7, and the formula to be monitored is $@_2 \boxdot (y \geq @_1 x)$. An example computation is shown in Figure 2.

There is only one formula to monitor with a single occurrence of an @ operator, namely $@_1 x$. Hence, the KNOWLEDGEVECTOR has a single entry which corresponds to p_1. Moreover, since the only remote expression to be tracked is x, $KV[1].values$ simply stores the value of x. In the figure, next to each event, we show $KV[1]$ at that instant for that process. $KV[1]$ is graphically displayed by a stack of two numbers, the top number showing $KV[1].seq$ and the bottom number showing the value for x.

The computation starts off with the initial values of $x = 5$ and $y = 7$. All processes know the initial value of x, hence the $KV[1].values$ for each process has value 5. It is easy to see that the monitored formula $\boxdot (y \geq @_1 x)$ holds initially at p_2. Subsequently, at p_1 there is an internal event e_{11} which sets $x = 9$ and updates $KV_1[1].values$ correspondingly. Process p_1 then sends a message to p_2 with a copy of its current KV. Another internal event e_{13} causes x to be set to 6. Process p_1 again sends a message, this time to p_3, with

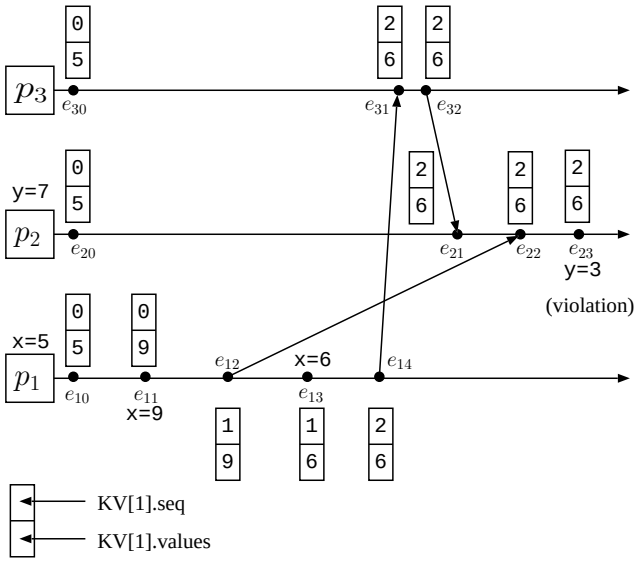

Fig. 2. Monitoring of $@_2\boxdot(y \geq @_1 x)$ at p_2

the updated KV. Process p_3 updates its KV and sends this on the message it sends to p_2.

At process p_2, the message sent by p_3 happens to arrive earlier than the message from p_1. Therefore, at event e_{21}, on receiving the message from p_3, process p_2 is able to update its KV to the one sent at event e_{14}. The monitor at p_2 again evaluates the property and finds that it still holds. The message sent by p_1 finally arrives at e_{22} but the KV piggybacked on is ignored as it has a smaller $KV[1].seq$ than $KV_2[1].seq$. The monitor correctly continues to declare the property valid. However, another internal event at p_2 causes the value of y to drop to 3, at which point the monitor detects a property violation.

7. THE DIANA TOOL

We have implemented the above technique as a tool, called DIANA (DIstributed ANAlysis). The architecture of DIANA is illustrated in Figure 3. DIANA is publicly available and can be downloaded from: `http://fsl.cs.uiuc.edu/diana/`. Both DIANA and the framework under which it operates are written in Java.

7.1 Actors

A number of formalisms can be used to reason about distributed systems, the most natural one being Actors [Agha 1986; Agha et al. 1997]. Actors are a model of distributed reactive objects and have a built-in notion of encapsulation and interaction, making them well suited to represent evolution and coordination between interacting components in distributed applications. Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors coordinate by asynchronously sending messages to each another. In the actor framework, a distributed system consists of different actors communicating through messages. Thus, there is an actor for each process in the system.

In the implementation, each type of actor (or process) is denoted by a Java class that

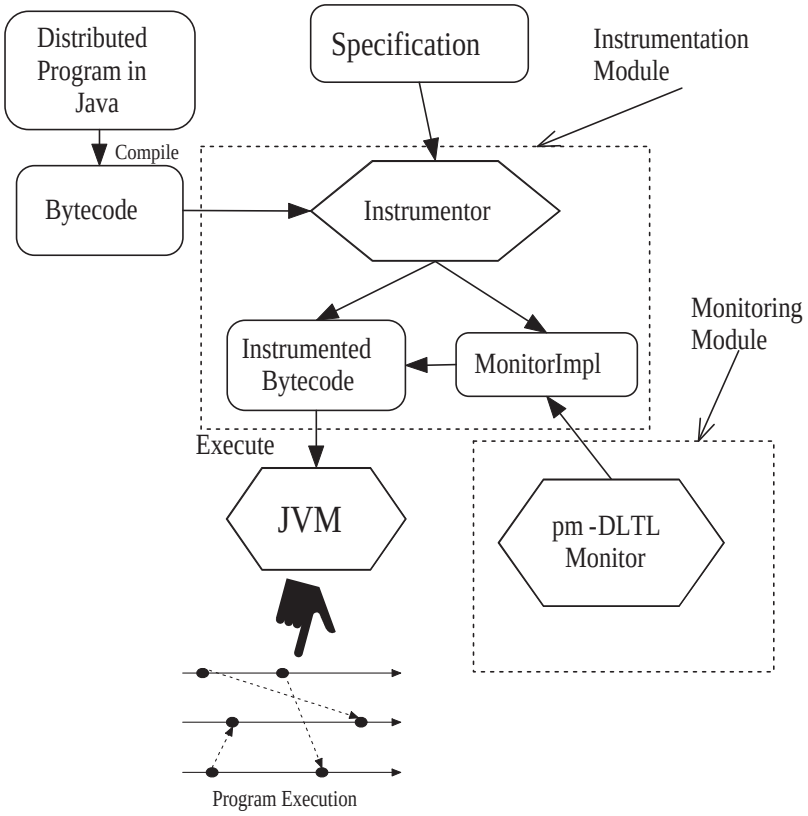

Fig. 3. The Architecture of DIANA

extends a base class `Actor`. This base class implements a message queue and provides the method `send` for asynchronous message sending. Each actor object executes as a separate process. The state of an actor is represented by the fields of the Java class. Each Java class also contains a set of `public` methods that can be invoked in response to messages received from other actors. A system level actor called *ActorManager* takes a message and transfers it to the message queue of the target actor. The target actor takes an available message from the message queue and invokes the method mentioned in the message. While processing a message, an actor may send messages to other actors. Message sending, being asynchronous, never blocks an actor. However, an actor blocks if there is no message in its message queue. An actor can create other actors by calling a special method `create` and pass the name of the actor to other actors through messages. We assume that each actor has a unique name, which is the name of the corresponding process. The name is passed as a string at the time creation of an actor. The system is initialized by the *ActorManager* object that creates all the actors in the system and starts the execution of the system.

7.2 Distributed Monitors in DIANA

The user of DIANA specifies a set of PT-DTL formulas to be monitored in a special file. With every formula, the user can also associate an *action*, which is a piece of Java code to

be invoked when the formula is violated.

As Figure 3 shows, the core of DIANA consists of two modules: an *instrumentation* module and a *monitoring* library. The instrumentation module takes the specification file and the distributed program written in the above framework and creates a Java class `MonitorImpl` that implements a local monitor for each actor (or process). The `MonitorImpl` class contains a field representing an array of knowledge vectors (one knowledge vector for each monitored formula) and a set of methods to update the knowledge vectors according to the KNOWLEDGEVECTOR algorithm. An instance of knowledge vector is constructed from the classes provided by the monitoring library. The instrumentation module automatically instruments the distributed program *at the bytecode level* (after compilation) to associate an instance of the class `MonitorImpl` with every actor at runtime. It also inserts code to every actor so that it invokes its local monitor (i.e. calls the appropriate methods of the instance of `MonitorImpl` class associated with it) whenever it modifies a field variable (internal event), sends a message, or invokes a method (receive event).

We handle an event corresponding to the creation of an actor in a special way. For every creation of new actor, the instrumentation tool inserts code to initialize the knowledge vectors of the newly created actor (child actor) with the knowledge vectors of the actor (parent actor) creating the new actor. This is because the events in the parent actor, before creating the new actor, causally precede any event in the child actor.

7.3 Test Cases

We implemented the following voting algorithm: a `Chair` process asks for vote on a resolution from N voters named $\texttt{Voter}_1$, $\texttt{Voter}_2$, ..., $\texttt{Voter}_N$, where N is initialized to an arbitrary but fixed positive number. We assume that the processes are connected in a tree shaped network with the `Chair` at the root of the tree and the voters at different nodes. Each voter randomly decides if it wants to vote for or against the resolution, and correspondingly stores 1 or 0 in a local state variable called `vote`. The voter then sends its decision to its immediate parent in the tree. The parent collects the votes and sends the sum of its vote and its progenies' votes to its immediate parent. The `Chair` process collects all the votes and rejects the resolution only if half or more voters have rejected. We monitor the following safety property at `Chair`:

$$@_{\texttt{Chair}}(\texttt{reject} \rightarrow (\texttt{sum}(@_{\{\texttt{Voter}_i | i \in [1..N]\}}(\texttt{vote})) < N/2))$$

The property was found to be violated in several runs: at some voter nodes, the voter sent the sum of its progenies' votes without adding its own vote. This resulted in the rejection of the resolution when it should have been accepted.

We have also tested a vector clock [Fidge 1988; Mattern 1989] algorithm implemented in the framework presented in this section. The algorithm was implemented as part of global snapshot and garbage collection algorithm. In this algorithm, each process is assumed to have a local vector clock V that it updates according to the standard vector clock algorithm [Fidge 1988] whenever there is an internal event, a send event or a receive event. The safety property that the algorithm must satisfy is that, at every process p_i: "all entries of the local vector clock must be greater than or equal to the local vector clock in a causally latest preceding state of any other process," expressed as the following i-formula:

$$@_i(\boxdot(V \geq \texttt{max}(@_{\{j|j\in[1..n]\}}V)))$$

where $V \geq V'$ when every entry in V is greater than or equal to the corresponding entry in V', and the function `max` takes a set of vectors as argument and returns a vector whose every entry is the maximum of the corresponding entries of the vectors in the set. Another safety property states that "at every process p_i the i-th entry in its local vector clock must be strictly greater than the i-th entry of the local vector clock of any other process". This can be expressed as the following i-formula:

$$@_i(\boxdot(V[i] > \texttt{max}(@_{\{j|j\in[1..n]\}}V[i])))$$

The second property was found to be violated in some computations due to a bug caused by failure to increment the i-th entry of the local vector clock of process p_i when receiving events.

These simple examples illustrate the practical utility and power of PT-DTL and the monitoring tool DIANA based on it.

8. RELATED WORK

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [Aumann 1976] and Halpern *et al.* [Fagin et al. 1995] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [Meenakshi and Ramanujam 2000] and develop methods for model checking formulas in this logic. Our communication primitive was in part inspired by this work, but we allow arbitrary expressions and atomic propositions over expressions in their logic.

Another closely related work is that of Penczek [Penczek 2000; Penczek and Ambroszkiewicz 1999] 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 for monitoring.

Leucker investigates linear temporal logic interpreted over restricted labeled partial orders called Mazurkiewicz traces [Leucker 2002]. An overview of distributed linear time temporal logics based on Mazurkiewicz traces is given by Thiagaranjan *et al.* in [Thiagarajan and Walukiewicz 1997]. [Alur et al. 1995] introduces 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 for the logic.

In recent years, there has been considerable interest in runtime verification [Havelund and Roşu 2004; Sokolsky and Viswanathan 2003]. Havelund *et al.* [Havelund and Roşu 2002] give algorithms for synthesizing efficient monitors for safety properties. Sen *et al.* [Sen et al. 2003] develop techniques for runtime safety analysis for multithreaded programs and introduce the tool JMPAX. Some other runtime verification systems include JPaX from NASA Ames [Havelund and Roşu 2001] and UPENN's Mac [Kim et al. 2001].

9. CONCLUSION AND FUTURE WORK

This work represents the first step in effective distributed monitoring. The work presented here suggests a number of problems that require further research. The logic itself could be made more expressive so that it expresses not only safety, but also liveness properties. One difficulty is that software developers are reluctant to use formal notations. A partial solution may be to merge the present work with a more expressive and programmer friendly monitoring temporal logic such as EAGLE [Barringer et al. 2004]. A complementary approach is to develop visual notations and synthesizing temporal logic formulas from such notations. There may also be the possibility of learning formulas based on representative scenarios.

An interesting avenue of future investigation that our work suggests is what we call *Knowledge-based Aspect-Oriented Programming*. Knowledge-based Aspect-Oriented Programming is a meta-programming discipline that is suitable for distributed applications. In this programming paradigm, appropriate actions are associated with each safety formula; these actions are taken whenever the formula is violated to guide the program and avoid catastrophic failures.

Acknowledgements

The first three authors are supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract F30602-00-2-0586, the DARPA IXO NEST Program, contract F33615-01-C-1907), the ONR Grant N00014-02-1-0715, and the Motorola Grant MOTOROLA RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

REFERENCES

AGHA, G. 1986. *Actors: A Model of Concurrent Computation*. MIT Press.

AGHA, G., MASON, I. A., SMITH, S. F., AND TALCOTT, C. L. 1997. A foundation for actor computation. *Journal of Functional Programming 7*, 1–72.

ALUR, R., PELED, D., AND PENCZEK, W. 1995. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*. San Diego, California, 90–100.

AUMANN, R. 1976. Agreeing to disagree. *Annals of Statistics 4*, 6, 1236–1239.

BABAOĞLU, O. AND MARZULLO, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, S. Mullender, Ed. Addison-Wesley, 55–96.

BARRINGER, H., GOLDBERG, A., HAVELUND, K., AND SEN, K. 2004. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*. Lecture Notes in Computer Science, vol. 2937. Springer-Verlag, Venice, Italy, 44–57.

E. M. CLARKE, J., GRUMBERG, O., AND PELED, D. A. 1999. *Model checking*. MIT Press.

FAGIN, R., HALPERN, J., MOSES, Y., AND VARDI, M. 1995. *Reasoning about Knowledge*. MIT Press.

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

HAVELUND, K. AND ROŞU, G. 2001. Java pathexplorer – A runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*. Montreal, Canada, June 18 - 21.

HAVELUND, K. AND ROŞU, G. 2001, 2002, 2004. *Workshops on Runtime Verification (RV'01, RV'02, RV'04)*. ENTCS, vol. 55, 70(4), to appear. Elsevier.

HAVELUND, K. AND ROŞU, G. 2002. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*. Lecture Notes in Computer Science, vol. 2280. Springer-Verlag, 342–356.

KIM, M., KANNAN, S., LEE, I., AND SOKOLSKY, O. 2001. Java-mac: a run-time assurance tool for java. In *Proceedings of Runtime Verification (RV'01)*. Electronic Notes in Theoretical Computer Science, vol. 55. Elsevier Science.

LEUCKER, M. 2002. Logics for mazurkiewicz traces. Tech. Rep. AIB-2002-10, RWTH, Aachen, Germany. April.

MANNA, Z. AND PNUELI, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York.

MANNA, Z. AND PNUELI, A. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York.

MATTERN, F. 1989. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. C. et. al., Ed. Elsevier Science, 215–226.

MEENAKSHI, B. AND RAMANUJAM, R. 2000. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*. Lecture Notes in Computer Science, vol. 1853. Springer-Verlag, 487–498.

PENCZEK, W. 2000. A temporal approach to causal knowledge. *Logic Journal of the IGPL 8*, 1, 87–99.

PENCZEK, W. AND AMBROSZKIEWICZ, S. 1999. Model checking of causal knowledge formulas. In *Workshop on Distributed Systems (WDS'99)*. Electronic Notes in Theoretical Computer Science, vol. 28. Elsevier Science.

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

SEN, K., ROŞU, G., AND AGHA, G. 2003. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC'03)*. Helsinki, Finland.

SEN, K., VARDHAN, A., AGHA, G., , AND ROŞU, G. 2004a. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*. IEEE, Edinburgh, UK, 418–427.

SEN, K., VARDHAN, A., AGHA, G., , AND ROŞU, G. 2004b. On specifying and monitoring epistemic properties of distributed systems. In *2nd International Workshop on Dynamic Analysis (WODA'04), Satellite workshop of ICSE 2004*. British Institution of Electrical Engineers (IEE), Edinburgh, UK, 32–35.

SOKOLSKY, O. AND VISWANATHAN, M. 2003. *Runtime Verification 2003*. Electronic Notes in Theoretical Computer Science, vol. 89. Elsevier Science. Proceedings of a *Computer Aided Verification (CAV'03)* satellite workshop.

THIAGARAJAN, P. S. AND WALUKIEWICZ, I. 1997. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Twelth Annual IEEE Symposium on Logic in Computer Science (LICS'97)*. Warsaw, Poland, 183–194.