On Specifying and Monitoring Epistemic Properties of Distributed Systems

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

We present an epistemic temporal logic which is suitable for expressing safety requirements of distributed systems and whose formulae can be efficiently monitored at runtime. The monitoring algorithm, whose underlying mechanism is based on symbolic knowledge vectors*, is distributed, decentralized and does not require any messages to be sent solely for monitoring purposes. These important features of our approach make it practical and feasible even in the context of large scale open distributed systems.*

1. Introduction

The discovery and prevention of software errors is a difficult problem involving many different aspects, such as incorrect or incomplete specifications, errors in coding, faults and failures in the hardware, operating system or network. Two prominent formal approaches used in checking for errors are: theorem proving and model checking. Theorem proving is powerful but labor-intensive, requiring intervention by someone with fairly sophisticated mathematical training. On the other hand, model checking is more of a push-button technology, but despite exciting recent advances, the size of systems for which it is feasible remains rather limited. As a result, most system builders continue to rely on testing to identify bugs in their implementation.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: it requires the software developer to translate properties into specific checks on the program state. Second, test coverage is rather limited. To mitigate the first problem, software often includes dynamic checks on the systems state to identify problems at run-time. Recently, there has been some interest in run-time monitoring techniques [1] which provide a little more rigor in testing. In this approach, monitors are automatically synthesized from a formal specification. 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.

In [6] we argue that distributed systems may be effectively monitored against formally specified safety requirements. 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 introduced an epistemic temporal logic for distributed knowledge, called *past time linear temporal logic* and abbreviated PT-DTL, and showed how monitors can be synthesized for it. PT-DTL formulae are local to particular processes and are interpreted over projections of global state traces that the current process *is aware of*. In this paper, we increase the expressiveness of PT-DTL and make it more programmer friendly by adding constructs similar to value binding in programming languages and quantification in first order logic. These constructs allow us to succinctly specify properties of open distributed systems involving data. The new logic is called XDTL and its novel features are inspired from EAGLE [3].

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 instance, a monitors "if a has received a value from b 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". Using XDTL, all one needs to do is to provide the safety policy as a formula:

$$\begin{array}{l}\texttt{valueReceived} \rightarrow \\ \quad @_b(\Diamond(\texttt{valueComputed} \wedge @_a(\Diamond\texttt{valueRequested})))\end{array}$$

$@$ is an *epistemic operator* and should be read "at"; $@_bF$ is a *remote property* that should be thought of as the value of F in the most recent local state of b that the current process is aware of. In PT-DTL[6], $@$ can only take one process as a subscript. In XDTL, as described later in the paper, $@$ can take any set of processes as a subscript together with a universal or an existential quantifier, so $@_b$ becomes "syntactic sugar" for $@_{\forall\{b\}}$ (or for $@_{\exists\{b\}}$). $\Diamond$ should be read "eventually in the past". Monitoring the formula above will involve sending no additional messages but only a few bits of information piggybacked on the messages already being passed for the computation.

Suppose that we want to restrict the above safety policy by imposing a further condition that the value received by a must be same as the value computed by b. To express this stronger property, we need to compare values in states at two process that are not directly related. This property cannot be directly expressed in PT-DTL without introducing extra variables in the program itself. However, adding extra variables in the program can potentially result in side-effects which are not desirable. An elegant way to solve the problem is to introduce the notion data-binding in the logic used for monitoring. Informally, we can restate the property as follows: a monitors "if a has received a value from b then remember the value received in a variable k and it must be the case that previously in the past at b the following held: b has computed the value and the computed value is equal to k and at a a request was made for that value in the past". This can be written formally as follows:

$$\begin{array}{l}\texttt{valueReceived} \rightarrow \ \mathsf{let}\ k = \texttt{value}\ \mathsf{in} \\ \quad @_b(\Diamond(\texttt{computedValue} \wedge (k = \texttt{valueComputed}) \\ \qquad \wedge @_a(\Diamond\texttt{requestedValue})))\end{array}$$

Informally, the construct "let $\vec{k} = \vec{\xi}$ in F" binds the value of the expressions $\vec{\xi}$ at process a with the logic variables $\vec{k}$ which can be referred by any expression in the formula F.

Another example in [6] regards monitoring certain 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 namely 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:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow (\texttt{state} = \texttt{leader} \rightarrow \\ \quad (@_b(\texttt{state} \neq \texttt{leader}) \wedge @_c(\texttt{state} \neq \texttt{leader}))\end{array}$$

We can write similar formulae 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.

However, the above formula does not specify the requirement that every process must know the name of the process that has been elected as leader. We cannot express this stronger requirement in PT-DTL. However, using the construct "let _ in _" and assuming that the variable `leaderName` contains the name of the leader, the requirement can easily be stated in XDTL as follows:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow \mathsf{let}\ k = \texttt{leaderName}\ \mathsf{in} \\ \quad (@_b(\texttt{leaderName} = k) \wedge @_c(\texttt{leaderName} = k))\end{array}$$

Note that the above formula assumes that the name of every process involved in leader election is known to us beforehand. 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 open distributed system where we 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 $i \neq a$ and use this set as a subscript to the operator @:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow \mathsf{let}\ k = \texttt{leaderName}\ \mathsf{in} \\ \quad @_{\forall\{i|i\neq a\}}(\texttt{leaderName} = k)\end{array}$$

$@_{\forall\{i|i\neq a\}}(\texttt{leaderName} = k)$ denotes the fact that the formula $\texttt{leaderName} = k$ must hold true at all processes i satisfying the predicate $i \neq a$. This is equivalent to the first order logic formula $\forall i\ .\ ((i \neq a) \rightarrow @_i(\texttt{leaderName} = k))$.

The logic XDTL proposed in this paper, extending PT-DTL with the construct "let _ in _" and with quantified sets of processes in the subscript of the epistemic operator @, is more expressive and elegant than PT-DTL. These benefits are attained without sacrificing efficiency and the decentralized nature of monitoring.

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [2] and Halpern *et al.* [4] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [5] and develop methods for model checking formulae in this logic. However, in our work we address the problem of monitoring and investigate an expressive distributed temporal logic that can be monitored in a decentralized way.

The rest of the paper is organized as follows. Section 2 describes the basic concepts of distributed systems. Section 3 introduces the more expressive PT-DTL which we call XDTL. In Section 4 we conclude by briefly sketching a decentralized monitoring algorithm.

2. Distributed Systems

We consider a distributed system as a collection of processes, each having a unique name and a local state, communicating with each other through asynchronous message exchange. The computation of each process is abstracted out in terms of *events* which can be of three types: *internal*, an event denoting local state update of a process, *send*, an event denoting the sending of a message by a process to another process, and *receive*, an event denoting the reception of a message by a process. Let E_i denote the set of events of process i and let E denote $\bigcup_i E_i$. Also, let $\lessdot \subseteq E \times E$ be 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 among the events in different processes and gives an abstraction of the *distributed computation* denoted by $\mathcal{C} = (E, \prec)$. In what follows, we assume an arbitrary but fixed distributed computation $\mathcal{C}$. Let us define $\preccurlyeq$ as the reflexive and transitive closure of $\lessdot$. In Fig. 1, $e_{11} \lessdot e_{23}$ and therefore also $e_{11} \prec e_{23}$. However, even though $e_{12} \not\lessdot e_{23}$, we have $e_{12} \prec e_{23}$ as process 2 gets a message from process 3 which contains knowledge of e_{12}.

The *local state* of a process is abstracted out in terms of a set of events. 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 process i when the event e has just occurred.

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 use the symbols s_i, s_i', s_i'' and so on to represent the local states of process i. We also assume that each local state s_i of each process 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 i.

We use the notation $causal_j(s_i)$ to refer to the latest state of process j of which process i knows while in state s_i. Formally, $causal_j(s_i) = s_j$ where s_j is a state at process j such that $s_j \preccurlyeq s_i$ and for all states s_j' in process j with $s_j' \preccurlyeq s_i$ we have $s_j' \preccurlyeq s_j$. For example, in Figure 1 $causal_1(\downarrow\!e_{23}) = \downarrow\!e_{12}$. Note that if $i = j$ then $causal_j(s_i) = s_i$.

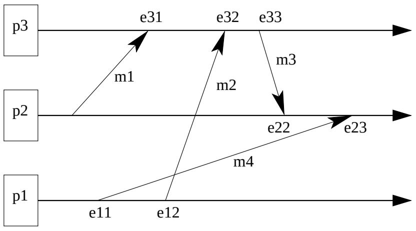

Figure 1. Sample Distributed Computation

3. Extended Distributed Temporal Logic

In order to reason about the global distributed computation locally, XDTL has a set of three new variants of *epistemic operators*, whose role is to evaluate an expression or a formula in the *last known state* of a remote process. We call such an expression or a formula *remote*. In addition to the epistemic operators, we add the construct "let $\vec{k} = \vec{\xi}$ in F" to XDTL to bind expressions to local logic variables that can be referred by any expression or formula in F.

The intuition underlying XDTL is that each process may be associated a local formula which, due to the epistemic operators, can refer to the global state of the distributed system. These formulae are required to be valid at the respective processes during a distributed computation. The distributed computation satisfies the specification when all the local formulae are shown to satisfy the computation. Next, we formally describe the syntax and semantics of XDTL.

3.1. Syntax

In the sequel, whenever we talk about an XDTL formula, it is in the context of a particular process, having the name i. We call such formulae *i-formulae* and let F_i, F_i', etc., denote them. Additionally, we introduce the notion of expressions local to a process i called as *i-expressions* and let ξ_i, ξ_i', etc., denote them. Informally, an i-expression is an expression over the global state of the system that process i is currently aware of. Local predicates on i-expressions form the atomic propositions on which the temporal i-formulae 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 $\{j \mid client(j)\}$. The following gives the formal syntax of XDTL with respect to a process i, where i and j are the name of any process (not necessarily distinct):

$F_i ::=$	$\text{true} \mid \text{false} \mid P(\vec{\xi_i}) \mid \neg F_i \mid F_i \ op\ F_i$	propositional
	$\mid \odot F_i \mid \Diamond F_i \mid \boxdot F_i \mid F_i\ \mathcal{S}\ F_i$	temporal
	$\mid @_{\forall J}F_j \mid @_{\exists J}F_j$	epistemic
	$\mid \text{let } \vec{k} = \vec{\xi_i} \text{ in } F_i$	binding
$\xi_i ::=$	$c \mid v_i \mid k \mid f(\vec{\xi_i})$	functional
	$\mid @_J\xi_j$	epistemic
$\vec{\xi_i} ::=$	$(\xi_i, \ldots, \xi_i)$	

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 on process 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 the set V_i containing all the local state variables of process i. c stays for constants, e.g., 0, 1, 3.14.

3.2. Semantics

The semantics of XDTL extends the semantics of PT-DTL by defining the three variants of epistemic operators

$$
\begin{array}{lll}
\mathcal{C}, s_i, [e] \models @_{\forall J} F_j & \text{iff } \forall j \,.\, (j \in J) \rightarrow \mathcal{C}, s_j, [e] \models F_j \text{ where } s_j = causal_j(s_i) \\
\mathcal{C}, s_i, [e] \models @_{\exists J} F_j & \text{iff } \exists j \,.\, (j \in J) \wedge \mathcal{C}, s_j, [e] \models F_j \text{ where } s_j = causal_j(s_i) \\
\mathcal{C}, s_i, [e] \models \mathsf{let}\ (k, \ldots, k') = (\xi_i, \ldots, \xi'_i)\ \mathsf{in}\ F_i & \text{iff } \mathcal{C}, s_i, [e, k \mapsto (\mathcal{C}, s_i, [e])[\![\xi_i]\!], \ldots, k' \mapsto (\mathcal{C}, s_i, [e])[\![\xi'_i]\!]] \models F_i \\
\\
(\mathcal{C}, s_i, [e, k \mapsto val])[\![k]\!] & = val \\
(\mathcal{C}, s_i, [e])[\![@_J \xi_j]\!] & = \{(\mathcal{C}, s_j, [e])[\![\xi_j]\!] \mid s_j = causal_j(s_i) \wedge j \in J\}
\end{array}
$$

Table 1. Semantics of XDTL

and the binding operator. The semantics is given by recursively defining the satisfaction relation $\mathcal{C}, s_i, [e] \models F_i$, where $[e]$ is an environment carrying the bindings for different logic variables which gets introduced by the "let _ in _" operator. $(\mathcal{C}, s_i, [e])[\![\xi_i]\!]$ is the value of the expression ξ_i in the state s_i under the environment $[e]$. Table 1 formally gives the semantics of the new operators of XDTL. For the semantics of other operators the readers are referred to [6]. We assume that expressions are properly typed. Typically these types would be `integer, real, strings`, etc. We also assume that $s_i, s'_i, s''_i, \ldots$ are states of process i and $s_j, s'_j, s''_j, \ldots$ are states of process j.

4. Monitoring Algorithm

To monitor XDTL formulae in a decentralized way, we synthesize *distributed monitors* as follows. For each process there is a separate monitor, called a *local monitor*, which checks the local XDTL formulae and can attach additional information to any outgoing message. This information can subsequently be extracted by the local monitor on the receiving side without changing the underlying semantics of the distributed program. The local monitor of each process i maintains a KNOWLEDGEVECTOR data-structure KV_i, storing for each process j in the system the status of all the safety policy sub-formulae and sub-expressions referring to j that i is aware of. The knowledge vector KV_i is appended to any message sent by i. When performing an internal computation step, the status of the local formulae and expressions is automatically updated in the local knowledge vector. When receiving a message from another process, the knowledge vector is updated if the received message contains more recent knowledge about any process in the system. To do this, a sequence number needs also to be maintained for each process in the knowledge vector. Unlike [6], the entries of KNOWLEDGEVECTOR are symbolic expressions instead of values. This is due to the fact that all the logic variables referred in an expression or a formulae may not be available at the time of evaluation of the expression or the formula. Therefore, the evaluation of a formula or an expression may be partial, containing the various logic variables. The logic variables in these formulae or expressions are replaced by actual values once they become available. A detailed discussion of the algorithm is beyond the scope of this short paper. However, readers are referred to [6, 3] for some of the similar ideas.

5. Conclusion

We believe that the logic XDTL presented in this paper is a powerful underlying specification formalism for distributed systems. Specifications expressed as XDTL formulae can be effectively monitored, even in the context of large scale open distributed systems. However, it is worthwhile to investigate other extensions that increase its expressiveness without sacrificing the efficiency of monitoring.

Acknowledgements

The first three authors are supported in part by 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 RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

References

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

[2] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6), 1976.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*.

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

[5] 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 *LNCS*.

[6] 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) (To Appear)*.