Efficient Decentralized Monitoring of Safety in Distributed Systems

Authors: Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Rošu

Animesh Jain
Poorvasha Dhananjay Chauhan

April 27, 2025

1 Introduction

The goal of this work is to ensure safety in distributed systems through monitoring. However, this poses several challenges, such as

1. the lack of a global clock,
2. asynchronous message passing,
3. and the need for local and efficient monitoring.

To address these challenges, the authors propose the use of Past-Time Distributed Temporal Logic (PT-DTL), a variant of past-time linear temporal logic.

1. Challenge with LTL: Monitoring with LTL requires collecting consistent global snapshots and exploring all possible interleavings of events, which is computationally expensive and impractical in large-scale distributed systems—even with techniques like partial order reduction.
2. PT-DTL Advantage: PT-DTL overcomes this by enabling each node to locally monitor properties using only its causal knowledge, eliminating the need for global snapshots or exhaustive interleaving checks.

2 Distributed Systems Model

2.1 Introduction

This report discusses a distributed system composed of n processes, $p_1, p_2, ..., p_n$. Each process p_i has its own local state s_i, which evolves as events occur, and the ability to communicate asynchronously with other processes by sending and receiving messages.

2.2 Events and Partial Orders

The computation of each process is abstracted out in terms of events, which can be of three types:

1. Internal events – actions that affect only the local state of a process
2. Send events – when a process sends a message to another
3. Receive events – when a process receives a message sent by another

Since there's no global clock, i.e., processes do not share a common notion of time, and events are only partially ordered, we define the following:
Let $E = \bigcup_{i=1}^{n} E_i$ be the union of all event sets from each process, where E_i is the set of events on process p_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. The structure described by C = (E,$\prec$) is called a distributed computation.
We also define $\preceq$ as the reflexive and transitive closure of $\lessdot$.

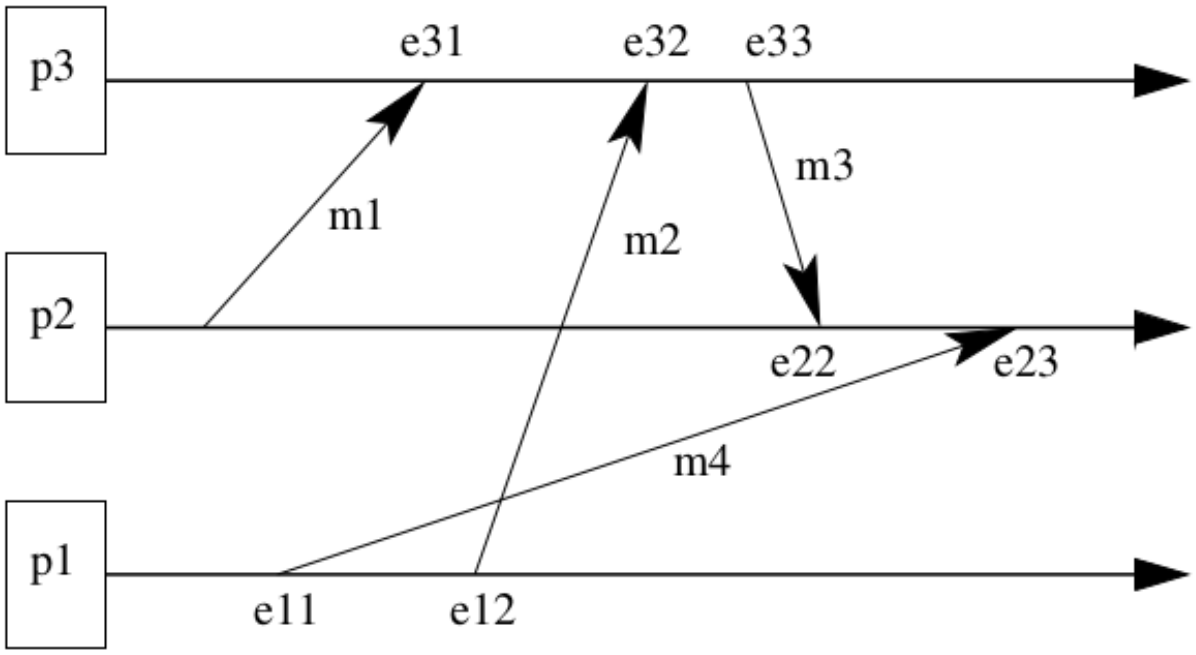

Figure 1: Sample Distributed Computation

2.3 Local State

The local state of a process is abstracted out in terms of a set of events. For e ∈ E, we define $\downarrow e = e`|e` \preceq e$, that is, $\downarrow e$ is the set of events that causally precede e. For e ∈ E_i, we can think of $\downarrow e$ as the local state of pi when the event e has just occurred.

We extend the definitions of $\lessdot$, $\prec$, and $\preceq$ to local states as follows:

$$\begin{aligned} \downarrow e \lessdot \downarrow e' &\iff e \lessdot e', \\ \downarrow e \prec \downarrow e' &\iff e \prec e', \\ \downarrow e \preceq \downarrow e' &\iff e \preceq e'. \end{aligned}$$

We denote the set of local states of a process p_i by $LS_i = \downarrow e | e \in E_i$ and let $LS = \bigcup_i LS_i$. Each local state s_i of each process p_i associates values to some local variables V_i, and $s_i(v)$ denotes the value of a variable $v \in Vi$ in the local state s_i at process p_i.

2.4 Causal

if $causal_j(s_i) = s_j$, then $s_j \in LS_j$ and $s_j \preceq s_i$, and for all $s_j` \in LS_j$, if $s_j` \preceq s_i$ then $s_j` \preceq s_j$. For example, in the sample distributed computation, $causal_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if i = j, then $causal_j(s_i) = s_i$.

The figure above shows a sample distributed computation, where we can see that $e_{11} \prec e_{23}$, $e_{12} \prec e_{23}$, and e11 $\lessdot$ e23. However, e12 $\lessdot$ e23.

3 PT-LTL

Past-time Linear Temporal Logic (PT-LTL) has been successfully used in the past to express, monitor, and predict violations of safety properties in software systems.

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

Figure 2: *PT_LTL_Semantics*

$$
\begin{array}{rl}
F ::= true | a \in A | \neg F | F op F & \textit{propositional} \\
| \bigcirc^{-1} F | \diamondsuit^{-1} F | \Box^{-1} F | F S F & \textit{temporal}
\end{array}
$$

Where *op* are the standard binary operators, namely $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and
$\bigcirc^{-1} F$ should be read as "previously",
$\diamondsuit^{-1} F$ as "eventually in the past",
$\Box^{-1} F$ as "always in the past", and
$F_1 \mathcal{S} F_2$ as "F1 since F2".

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 Figure 2.

For ex, the formula $\Box^{-1}((\text{action} \wedge \bigcirc^{-1} \neg \text{action}) \rightarrow (\neg \text{stop} \mathcal{S} \text{start}))$
states that "whenever action starts to be true, it is the case that start was true at some moment in the past and since then stop was never true", or in other words that the action is taken only when the system is active.

4 PT-DTL - A Distributed Temporal Logic

4.1 Overview

PT-DTL (Partially Timed Distributed Temporal Logic) is an extension of PT-LTL (Partially Timed Linear Temporal Logic) that allows for local monitoring with knowledge of remote states. It introduces epistemic operators ($@_j$) that enable processes to refer to remote variables and formulae, enabling them to monitor global properties locally.

4.2 PT-DTL Syntax

PT-DTL consists of two main components:

i-formulae (F_i): Formulae local to process p_i.

i-expressions (ξ_i): Expressions that can include local or remote expressions.

The epistemic operators in PT-DTL are:

$@_j F_j$: A formula about process p_j that is known to process p_i.

$@_j \xi_j$: The value of expression ξ_j on process p_j as known to process p_i.

4.3 PT-DTL Semantics

PT-DTL semantics are defined over causal states rather than linear traces. For a configuration C and local state s_i of process p_i, the semantics are defined as follows:

$(C, s_i)[[@_j \xi_j]] = \xi_j$ at $(causal_j(s_i))$, where ξ_j is an expression from process p_j, $@_j \xi_j$ is the value of ξ_j at process p_j, as known to process p_i, and $causal_j(s_i)$ is the most recent state of p_j known to p_i at its local state s_i.

This approach enables processes to make decisions based on what they know about other processes, avoiding the need for global synchronization and linearization. It captures remote state awareness efficiently and forms the basis for local monitoring of global safety properties.

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

$$
\begin{array}{ll}
(\mathcal{C}, s_i)[\![v_i]\!] & = s_i(v_i), \text{ that is, the value of } v_i \text{ in } s_i \\
(\mathcal{C}, s_i)[\![c_i]\!] & = c_i \\
(\mathcal{C}, s_i)[\![f(\xi_i, \ldots, \xi_i')]\!] & = f((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!]) \\
(\mathcal{C}, s_i)[\![@_j \xi_j]\!] & = (\mathcal{C}, s_j)[\![\xi_j]\!] \text{ where } s_j = causal_j(s_i)
\end{array}
$$

Figure 3: Semantics of PT-DTL

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

Figure 4: Recursive Semantics of PT-DTL

4.4 PT-DTL Examples

Leader Election: "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."

Property: Only one leader Formula:

$$leaderElected \rightarrow (state = leader \rightarrow \bigwedge_{j \neq i} (@_j(state \neq leader))$$

Voting: "If the resolution is accepted, then more than half of the processes say yes."

Property: Majority votes yes Formula:

$$accepted \rightarrow (@_1(vote) + ... + @_n(vote)) > n/2$$

Server Reboot: "The server accepts to reboot only after knowing that each client is inactive and aware of the warning to reboot."

Property: Reboot only if all clients are inactive and know about warning Formula:

$$rebootAccepted \rightarrow \bigwedge_{client} (@_{client}(inactive \wedge @_{server} rebootWarning))$$

Request and Reply: "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." Formula:

$$\text{receivedValue} \rightarrow @_b(\Diamond^{-1}(\text{computedValue} \wedge @_a(\Diamond^{-1}\text{requestedValue})))$$

Temperature Alarm: "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." 5. **Temperature Alarm**: "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." Formula:

$$\text{alarm} \rightarrow \Diamond^{-1}((\text{myTemp} - @_b\text{otherTemp}) > \text{allowed})$$

Airplane Landing: "If my airplane is landing, then the runway assigned by the airport matches the one that I plan to use." Formula:

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

These examples demonstrate how PT-DTL can be used to express and monitor various global properties in a distributed system, without the need for global synchronization or linearization.

The figures above illustrate the semantics of PT-DTL, showing how the epistemic operators are used to refer to remote states and formulae.

5 Monitoring Algorithm for PT-DTL

In this section, we describe an automated technique to synthesize efficient distributed monitors for safety properties in distributed systems expressed in PT-DTL. The synthesized monitor is distributed, in the sense that it consists of separate, local monitors running on each process. A local monitor can attach additional information to any 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.

5.1 Key Guiding Principles

The key guiding principles in the design of this technique are:

1. The local monitors should be fast, so that monitoring can be done online.
2. The local monitors should have little memory overhead; in particular, they should not need to store the entire history of events on a process since this can be quite large.
3. Additional messages needed to be sent between processes solely for the purpose of monitoring should be minimized and ideally should be zero. Further, additional information piggybacked on regular messages (those generated as part of the distributed computation) should be small.

5.2 Monitoring Algorithm Overview

The monitoring algorithm for PT-DTL aims to achieve these goals by leveraging the causal structure of the distributed computation and the semantics of the PT-DTL logic. The algorithm works as follows:

1. For each process p_i, the local monitor maintains a summary of the local state of p_i and the causal information about the local states of other processes that p_i is aware of.
2. Whenever a new event occurs on p_i, the local monitor updates its summary and checks if the PT-DTL formula is satisfied at the current local state.
3. If the formula is not satisfied, the local monitor can take appropriate action, such as raising an alarm or triggering a recovery mechanism.
4. The local monitors communicate with each other by piggybacking small amounts of information on regular messages exchanged between the processes. This allows them to keep their local summaries up-to-date without the need for additional messages.

5.3 Advantages of the Approach

The key advantage of this approach is that it allows for efficient online monitoring of global properties in a distributed system, without the need for global synchronization or the storage of the entire history of events. The local monitors can make decisions based on their current local state and the causal information they have about the remote states, ensuring that the monitoring can be done in a scalable and efficient manner.

6 Knowledge Vectors

The key guiding principles in the design of this technique are:

1. The local monitors should be fast, so that monitoring can be done online.
2. The local monitors should have little memory overhead, in particular, it should not need to store the entire history of events on a process since this can be quite large.
3. Additional messages needed to be sent between processes solely for the purpose of monitoring should be minimized and ideally, should be zero. Further, additional information piggybacked on regular messages (those generated as part of the distributed computation) should be small.

6.1 Knowledge Vectors, Motivation

Consider evaluating a remote j-expression $@_j\xi_j$ at process p_i. The naive solution is that process p_j simply piggybacks the value of ξ_j evaluated at p_j, with every message that it sends out. The recipient process p_i can extract this value and use it as the value of $@_j\xi_j$. The problem with this approach is that it adds more overhead to each message sent, and more importantly, the messages from p_j could reach p_i in arbitrary order.

To keep track of the causal history, or in other words the most recent knowledge, we also need the event number at p_j at which these expressions were sent out in messages so that stale information in a reordered message sequence can be discarded.

6.2 Knowledge Vectors

Causal ordering can be effectively accomplished by using an array called KNOWLEDGE VECTOR with an entry for any process p_j for which there is an occurrence of $@_j$ in any PT-DTL formula at any process. Knowledge vectors are motivated and inspired by vector clocks.

The size of KNOWLEDGE VECTOR is not dependent on the number of processes but on the number of remote expressions and formulae. Let KV[j] denote the entry for process p_j on a vector KV. KV[j] contains the following fields:

1. The sequence number of the last event seen at p_j, denoted by KV[j].seq;
2. A set of values KV[j].values storing the values of j-expressions and j-formulae.

Each process p_i keeps a local KNOWLEDGE VECTOR 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 KVm.

6.3 Update Algorithm

The algorithm for the update of KNOWLEDGE VECTOR KV_i at process p_i is given below:

1. **[internal]:** Update $KV_i[i]$. For this, we evaluate $\text{eval}(\xi_i, s_i)$ and $\text{eval}(F_i, s_i)$ for each $@_i\xi_i$ and $@_iF_i$, respectively, and store them in the set $KV_i[i]$.values;
2. **[send m]:** $KV_i[i].\text{seq} \leftarrow KV_i[i].\text{seq} + 1$. Send KV_i with m as KV_m;
3. **[receive m]:** For all j, if $KVm[j].\text{seq} > KVi[j].\text{seq}$ then $KVi[j] \leftarrow KVm[j]$, that is, $KVi[j].\text{seq} \leftarrow KV_m[j].\text{seq}$, and $KV_i[j].\text{values} \leftarrow KV_m[j].\text{values}$.

6.4 Proposition

Proposition 1: For any process p_i and any j, the entry for ξ_j or F_j in KVi[j].values contains the value of $@_j\xi_j$ or $@_jF_j$, respectively.

$KV_i[j]$.value contains the latest values that p_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 $@_jF_j$, process p_i can just use the entry corresponding to ξ_j or Fj in the set KVi[j].values.

6.5 Notes

Note:

1. The sequence number needs to be incremented only when sending messages.
2. The algorithm aims to minimize local work when sending a message. However, it is important to note that:
   - The values calculated at step 1 are only needed when an outgoing message is generated at step 2.
   - One could evaluate all the expressions ξ_i and F_i at step 2, right before the message is sent out.
   - This approach would reduce the runtime overhead at step 1 but increase it at step 2.
   - Depending on the specific application, one may prefer one approach over the other.
3. The initial values for all the variables in the distributed program can be determined by:
   - A static analysis of the program.
   - A distributed broadcast at the beginning of the computation.
4. It is assumed that each process p_i has complete knowledge of the initial values of remote expressions for all processes. These values are then used to initialize the entries $\texttt{KV}_i[j].\texttt{values}$ in the KNOWLEDGE VECTOR of p_i for all j.

7 Monitoring a Local PT-DTL Formula

We now describe the details of a local monitor. The monitoring algorithm for a PT-DTL formula is similar in spirit to that for an ordinary PT-LTL formula described in [20]. The key differences and functionalities are as follows:

1. The monitoring algorithm allows for:
   - Remote expressions whose values need to be transferred from a remote process to the current process.
   - Remote formulae whose validity needs to be verified in the current process.
2. Once the `KNOWLEDGE VECTOR` is properly updated, the local monitor can:
   - Compute the boolean value of the formula to be monitored.
   - Recursively evaluate the boolean value of each of its subformulae in the current state.
3. The local monitor may also utilize:
   - The boolean values of subformulae evaluated in the previous state.
   - The values of remote expressions and remote formulae.
4. A function `eval` is defined to:
   - Take advantage of the recursive nature of the temporal operators, as described in Table 3.
   - Calculate the boolean value of a formula in the current state based on:
     - Its boolean value in the previous state.
     - The boolean value of its subformulae in the current state.
5. The function components include:
   - `op(Fi)`: Returns the operator of the formula F_i.
   - `binary(op(Fi))`: Returns `true` if `op(Fi)` is binary.
   - `unary(op(Fi))`: Returns `true` if `op(Fi)` is unary.
   - `left(Fi)`: Returns the left subformula of F_i.
   - `right(Fi)`: Returns the right subformula of F_i when `op(Fi)` is binary.
   - `subformula(Fi)`: Returns the subformula of F_i otherwise.
6. The variable `index` represents the index of a subformula.

8 Algorithms

The algorithms for evaluating formulae and expressions, as well as the initialization procedure, are shown in $F_i$gures 5, 6, and 7, respectively.

```
array now; array pre; int index;
boolean eval(Formula F_i, State s_i){
  if binary(op(F_i)) then{
    lval ← eval(left(F_i), s_i);
    rval ← eval(right(F_i), s_i);}
  else if unary(op(F_i)) then
    val ← eval(subformula(F_i), s_i);
  index ← 0;
  case(op(F_i)) of{
    true : return true;  false : return false;
    P(ξ_i⃗) : return P(eval(ξ_i, s_i), ..., eval(ξ'_i, s_i)));
    op :  return rval op lval; ¬ :  return not val;
    S : now[index] ← (pre[index] and lval) or rval;
      return now[index++];
    □ : now[index] ← pre[index] and val;
      return now[index++];
    ◇ : now[index] ← pre[index] or val;
      return now[index++];
    ⊙ : now[index] ← val; return pre[index++];
    @_j F_j : return value of F_j from KV_i[j].values;
  }
}
```

Figure 5: Algorithm 1 eval Formula

```
value eval(Expression ξ_i, State s_i){
  case(ξ_i) of{
    v_i: return s_i(v_i);  c_i: return c_i;
    f(ξ_i^1, ..., ξ_i^k): return f(eval(ξ_i^1, s_i), ..., eval(ξ_i^k, s_i));
    @_j ξ'_j: return value of ξ'_j from KV_i[j].values;
  }
}
```

Figure 6: Algorithm 2 eval Expression

```
boolean init(Formula F_i, State s_i){
  if binary(op(F_i)) then{
    lval ← init(left(F_i, s_i));
    rval ← init(right(F_i, s_i));}
  else if unary(op(F_i)) then
     val ← init(subformula(F_i, s_i));
  index ← 0;
  case(op(F_i)) of{
    true : return true;  false : return false;
    P(ξ_i⃗) : 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++];
  }
}
```

Figure 7: Algorithm 3 init

We use the function `eval` after every internal event or before sending any message to update the set KVi[i].values. We assign `now` to `pre` and, if a monitored PT-DTL formula F_i is specified for a process p_i, we designate p_i as the formula's owner.

At the owner process, we evaluate F_i using `eval` after each internal and receive event, following the update of the KNOWLEDGEVECTOR. If F_i evaluates to false, we report a violation warning.

The time and space complexity of this algorithm at every event is θ(m), where m is the size of the original local formula.

9 Example

The example computation is illustrated in Figure 8. Let us consider three processes, p_1, p_2, and p_3. The example illustrates the monitoring of a PT-DTL formula as follows:

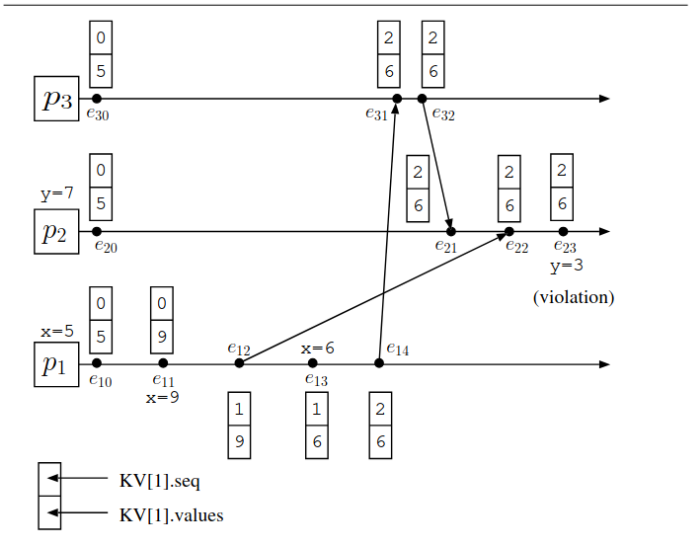

Figure 8: Monitoring of $\mathrm{F}(y \geq @_1 x)$ at p_2

1. **Initial Setup**:
   - Process p_1 has a local variable x with an initial value of 5.
   - Process p_2 has a local variable y with an initial value of 7.
   - Process p_2 monitors the formula $\Box^{-1}(y \geq @_1 x)$.
2. **Knowledge Vector**:
   - There is only one formula to monitor with a single occurrence of the `@` operator, namely $@_1 x$.
   - The **KNOWLEDGE VECTOR** has a single entry corresponding to p_1.
   - Since the only remote expression to be tracked is x, `KV[1].values` stores the value of x.
   - In the figure, `KV[1]` is graphically displayed by a stack of two numbers: the top number shows `KV[1].seq` and the bottom number shows the value for x.
3. **Computation Steps**:
   - The computation starts with initial values: $x = 5$ and $y = 7$.
   - All processes know the initial value of x, so `KV[1].values` for each process is 5.
   - The monitored formula $\Box^{-1}(y \geq @_1 x)$ holds initially at p_2.
4. **Events**:
   - An internal event e_{11} at p_1 sets $x = 9$ and updates `KV[1].values` accordingly.
   - Process p_1 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 sends another message to p_3 with the updated `KV`.
   - Process p_3 updates its `KV` and sends this on the message to p_2.

5. **Message Arrival**:
    - At p_2, the message from p_3 arrives before the message from p_1.
    - At event e_{21}, upon receiving the message from p_3, process p_2 updates its `KV` to the one sent at event e_{14}.
    - The monitor at p_2 evaluates the property and finds that it still holds.
    - The message from p_1 arrives at e_{22}, but the `KV` piggybacked on it is ignored due to a smaller `KV[1].seq` than `KV2[1].seq`.
    - The monitor continues to declare the property valid.
6. **Property Violation**:
    - An internal event at p_2 causes the value of y to drop to 3.
    - The monitor detects a property violation at this point.

10 Conclusion

In summary, this work presents an efficient decentralized monitoring algorithm for safety properties in distributed systems, using the PT-DTL formalism and knowledge vectors to track causal history and remote state information. The key contributions are the design of the monitoring algorithm, the use of knowledge vectors, and the formal guarantees provided by the approach. The proposed solution addresses the challenges of monitoring in distributed systems, such as the lack of a global clock and the need for local and efficient monitoring.