In Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08), St. Louis, MO, USA, Apr 2008

Real-Time Distributed Discrete-Event Execution with Fault Tolerance

Thomas Huining Feng and Edward A. Lee
Center for Hybrid and Embedded Software Systems
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{tfeng, eal}@eecs.berkeley.edu

Abstract

We build on PTIDES, a programming model for distributed embedded systems that uses discrete-event (DE) models as program specifications. PTIDES improves on distributed DE execution by allowing more concurrent event processing without backtracking.

This paper discusses the general execution strategy for PTIDES, and provides two feasible implementations. This execution strategy is then extended with tolerance for hardware errors. We take a program transformation approach to automatically enhance DE models with incremental checkpointing and state recovery functionality. Our fault tolerance mechanism is lightweight and has low overhead. It requires very little human intervention. We incorporate this mechanism into PTIDES for efficient execution of fault-tolerant real-time distributed DE systems.

1. Introduction

Large-scale distributed embedded systems are a mixture of software components, hardware devices (such as sensors and actuators), and networks. Examples include automotive and avionics systems. From a high-level design perspective, the discrete-event (DE) models of these systems consist of interconnected actors accepting inputs and producing outputs via the connections between them. Following DE semantics, the actors must process their input events in the order of the model time associated with those events. A strategy for executing distributed DE models must obey this ordering of events across computers in a network.

Traditional conservative execution strategies for distributed DE models, such as the one proposed by Chandy and Misra [3], require a computer to wait for events to be present in all its input queues before it can decide the next event to process. Optimistic approaches, such as Time Warp, proposed by Jefferson [9], allow a computer to process events without always making sure that the time order is conformed to. When the system detects violation of time order, the state of some actors needs to be recovered, and events need to be processed again in the correct order. The need for backtracking and the domino effect that backtracking may cause in a distributed system make it hard for the optimistic approaches to be applied to embedded systems.

To address the problems of existing approaches, PTIDES (Programming Temporally Integrated Distributed Embedded Systems) has been proposed as a strategy for executing distributed DE models in [14]. By taking advantage of static information about dependency between actors, the PTIDES execution strategy reduces the unnecessary wait for input events and improves concurrency without resorting to an optimistic approach that has the need for backtracking.

In this paper we present our work on a fault-tolerant extension to PTIDES, which supports selective backtracking for fault tolerance (not for performance gains, as in Time Warp). It is based on a program transformation that automatically enhances models with incremental checkpointing and state recovery functionality [8]. This extra functionality makes it possible for the actors to recover state that is invalidated by hardware errors. The difference between the state recovery in our approach and that in optimistic approaches to distributed DE execution is that the former is only performed in reaction to hardware errors, which are rare in practice, while the latter is a requirement to remedy the misordering of events in an optimistic execution.

Compared to other software fault tolerance techniques, such as object serialization and Enterprise JavaBeans (EJB) using a back-end database, the incremental checkpointing that we developed is more suitable for real-time distributed embedded systems for its low run-time overhead and small memory footprint. The automatic program transformation tool that we implemented also makes it easy to apply this technique to preexisting programs that do not have fault-tolerance capability.

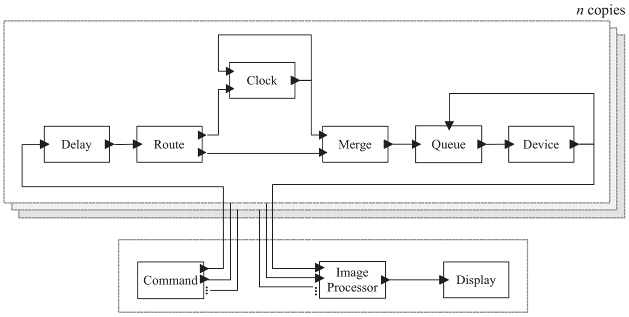


Figure 1. The camera application in PTIDES

The contribution of this paper is twofold. Firstly, we provide two feasible implementations for the general PTIDES execution strategy: one resembling traditional conservative strategies and the other utilizing time-synchronized platforms to provide better concurrency. Secondly, we extend the PTIDES execution strategy with a fault-tolerant feature that allows actors to automatically backtrack to previous state when hardware errors occur.

The following sections are arranged as follows: in Section 2, we motivate this work with a real-time distributed image composition application. The PTIDES execution strategy for DE is reviewed in Section 3, and then two feasible implementations are discussed. In Section 4, we present our incremental backtracking mechanism. In Section 5, we demonstrate the backtracking mechanism by extending the image composition application with it. The merit of our approach is assessed in Section 6 with a comparison to related work. Section 7 concludes the discussion.

2. Real-Time Image Composition Application

We motivate our work with the same experimental application considered in [14]. [1] The functionality of this application is to compose images periodically acquired from distributed cameras in real time. In the physical setup, n cameras are placed around a football field, all connected to a central computer via Ethernet. Figure 1 shows the simplified DE model. (Actors are shown as boxes, and the links between triangular ports represent communication channels for events.) The upper part of the figure shows the actors running on each camera. There are n identical copies of the upper part in the model. The lower part shows the actors running on the central computer.

The Command actor represents an input console that a human operator can use to issue two kinds of parametrized commands: "change frequency" that is used to change a camera's image-acquisition frequency, and "adjust zoom" that is used to adjust a camera's zoom. This actor assigns time tags to the commands before it sends them as output events. The time tags are in terms of model time (or virtual time), which are chosen such that they equal to the physical time (or real time) at which the commands are issued by the human operator. The Delay actor receives those commands via the network, and increases their time tags by a fixed amount. The Delay then immediately transmits the commands to the Route actor, which routes them to the designated receivers. The Clock accepts "change frequency" commands only, which instruct it to change the frequency at which it produces triggering events to its output. Each triggering event is also sent back to the Clock itself to trigger the next triggering event. The Merge actor merges the triggering events from the Clock and the "adjust zoom" commands from the Route in their time tag order, and sends them to the Queue. The Queue temporarily stores the commands until the Device is available to handle them. To handle a triggering event, the Device invokes a function in the camera's API (Application Programming Interface) to take a picture. An image is returned on success. To handle an "adjust zoom" command, the Device adjusts the camera's zoom by invoking another function in the API, which returns a success flag after the zoom is adjusted. If the operation in a functions fails, an error code is returned that describes the problem. When acquired, the images are sent back to the central computer via the same network. The Image Processor composes the images from all the cameras that are acquired at the same time to produce a composite view on the Display.

Two research challenges are highlighted in this prototypical application:

- *Timing*. A time tag is assigned to each event. The Command actor can be viewed as a special kind of sensor that receives external data. It associates with its output events time tags equal to the current physical time. The Device actor and the Display actor are actuators. The time tags of their input events represent the physical time at which those events should be processed by the underlying hardware. Obviously, if the events are received later than their time tags, then the actuators will not be able to actuate the hardware at the requested time, causing the problem of missing deadlines. To address this problem, the PTIDES execution strategy takes advantage of static information about dependency between actors to reduce the unnecessary delay in traditional conservative strategies.

- *Fault tolerance*. Sensors and actuators may generate errors. [2] We will not discuss the handling of those er-

[1] This application is inspired by the "eye vision" project (http://www.ri.cmu.edu/events/sb35/tksuperbowl.html) at CMU and CBS Television.

[2] In the scope of this paper, we do not consider design errors in the

rors, because it is application-specific. However, we will examine how hardware errors may affect event processing in software components. According to traditional conservative strategies, if the processing of an event depends on the successful completion of a hardware operation, then the actor cannot process the event until the operation finishes. We provide better concurrency by parallelizing the hardware operation and event processing. In the rare case where errors occur, the actor can recover its previous state with our backtracking mechanism.

3. Discrete Event Execution in PTIDES

A conservative strategy for executing distributed DE models proposed by Chandy and Misra [3] requires a computer to wait for events to be present in all its input queues before it can decide the next event to be processed. Based on the assumption that input queues receive events in time tag order, the next event to be processed is the one with the smallest time tag among the present events. The wait for events to be present in all input queues incurs unnecessary delay. An improvement is to have the computers generate "null messages" [12] from time to time, which inform their recipients to advance time even though there are no events for them. However, the extra null messages themselves are an overhead. Optimistic approaches, such as Time Warp [9], allow computers to process events out of order. Unfortunately, they require backtracking. This on the one hand complicates the implementation, and on the other hand prevents their application to embedded systems involving hardware devices that cannot backtrack once operations are performed.

In this section we first review the PTIDES execution strategy [14] for actor-oriented models. It improves upon the conservative strategies and provides more concurrency. We then outline two feasible implementations for PTIDES.

3.1. Relevant dependency

An actor $\alpha \in A$, where A is the set of all actors in the model, has a set of input ports I_α and a set of output ports O_α. Without loss of generality, in this section, we shall consider an actor's internal state as output to an output port, so we need not explicitly discuss internal state. We shall let $I = \bigcup_{\alpha \in A} I_\alpha$, $O = \bigcup_{\alpha \in A} O_\alpha$, and $P = I \cup O$.

During execution, each event e is associated with a *time tag* represented by $tag(e) \in \mathbb{R}^+$. It denotes the model time at which e was produced. The events that an actor receives

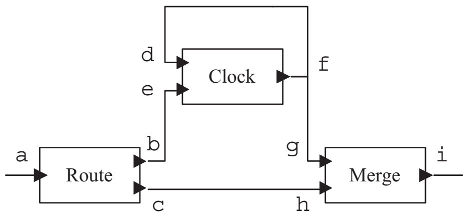


Figure 2. An example of minimum delay $\delta(\mathtt{a},\mathtt{i}) = \min\{\delta(\mathtt{a},\mathtt{c})+\delta(\mathtt{h},\mathtt{i}),\ \delta(\mathtt{a},\mathtt{b})+\delta(\mathtt{e},\mathtt{f})+ k \times \delta(\mathtt{d},\mathtt{f}) + \delta(\mathtt{g},\mathtt{i})\}$ **(**$k \geq 0$**)**

should be processed semantically in non-decreasing order [3] of their time tags. In particular, if distinct events e_1 and e_2 are received by the same actor, and if they have influence on the same output signal (i.e., the set of events output via a port over time), then e_1 must be processed prior to e_2 if $tag(e_1) < tag(e_2)$.

Before execution, PTIDES performs static analysis to obtain dependency information. The first step of computing dependency is to determine the minimum delay between pairs of ports, represented by a function $\delta : P \times P \to \mathbb{R}^+ \cup \{\infty\}$. $\delta(p_1, p_2)$ is defined as the minimum difference between $tag(e_1)$ and $tag(e_2)$ for any event e_1 at p_1 and e_2 at p_2 that depends on e_1. (We say e_2 depends on e_1 if at execution time, e_2's value or time tag may be totally or partially determined by e_1. We also say e_1 influences e_2 if e_2 depends on e_1.) We shall let $\delta(p_1, p_2) = \infty$ if the events at p_2 never depend on those at p_1. Note that the delay $\delta(p_1, p_2)$ is in terms of model time and does not depend on the physical time that it takes for events at p_1 to influence events at p_2.

Figure 2 shows the example of computing $\delta(\mathtt{a}, \mathtt{i})$, assuming that the minimum delay within individual actors is predefined, [4] and that the outputs of an actor always depend on the most recent inputs at all input ports. By applying min-plus algebra [1], $\delta(\mathtt{a}, \mathtt{i})$ is computed as the minimum delay of all the paths from a to i. A *path* from a to i is a sequence of ports $[p_1, p_2, \cdots, p_n]$ such that $p_1 = \mathtt{a}$, $p_n = \mathtt{i}$, and for every p_k where $1 < k \leq n$, either p_k is connected to p_{k-1}, or p_{k-1} and p_k are input port and output port of the same actor, respectively. The paths from a to i in this example include [a, c, h, i] and [a, b, e, f, g, i]. Because of the loop, there are an infinite number of paths, but these

software. A future extension is to adapt our fault-tolerance technique to software errors.

[3] Normally, we enhance each time tag with an integer to break the tie of events at the same model time, resulting in a time model known as superdense time [11]. For the purposes of this paper, this enhancement adds only complexity, so we omit it. Our results carry over trivially when using superdense time.

[4] The minimum delay within actors can be predefined as part of the actors' behavioral interface, which is not discussed here.

two have the smallest delay. The delay of [a, c, h, i] is computed by the sum of $\delta(\mathtt{a},\mathtt{c})$ and $\delta(\mathtt{h},\mathtt{i})$. The delay of $[\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f},\mathtt{g},\mathtt{i}]$ is the sum of $\delta(\mathtt{a},\mathtt{b})$, $\delta(\mathtt{e},\mathtt{f})$ and $\delta(\mathtt{g},\mathtt{i})$. (Direct connections do not cause any delay in model time.) The minimum of these is the minimum delay from a to i.

We then partition I into a set of equivalence classes $\mathbb{E} = \{\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_k\} \subseteq 2^I$. Distinct input ports $i_1 \in I$ and $i_2 \in I$ are in the same equivalence class, traditionally denoted by $i_1 \sim i_2$, if and only if they belong to the same actor, and there exists an output port o of that actor that satisfies $\delta(i_1, o) < \infty$ and $\delta(i_2, o) < \infty$. Intuitively, if $i_1 \sim i_2$, the events that they receive may influence the same output signal. Therefore, those events must be processed in the order of their time tags. However, if $i_1 \nsim i_2$, any event e_1 at i_1 and event e_2 at i_2 do not have influence on the same output signal, so the order in which e_1 and e_2 are processed is insignificant. To continue with the example in Figure 2, $\{\mathtt{a}\}$, $\{\mathtt{d},\mathtt{e}\}$ and $\{\mathtt{g},\mathtt{h}\}$ are equivalence classes in $\mathbb{E}$, due to the assumption that the outputs of an actor always depend on the most recent inputs at all input ports.

The *relevant dependency* $d : \mathbb{E} \times \mathbb{E} \to \mathbb{R}^+ \cup \{\infty\}$ is defined as:

$$d(\mathcal{E}_1, \mathcal{E}_2) = \min_{i_1 \in \mathcal{E}_1, i_2 \in \mathcal{E}_2} \{\delta(i_1, i_2)\}$$

For equivalence classes $\mathcal{E}_1 \in \mathbb{E}$ and $\mathcal{E}_2 \in \mathbb{E}$, $d(\mathcal{E}_1, \mathcal{E}_2)$ represents the minimum delay in model time that it takes for any event at a port in $\mathcal{E}_1$ to influence an event at a port in $\mathcal{E}_2$.

For example, in Figure 2, $d\big(\{\mathtt{a}\}, \{\mathtt{d},\mathtt{e}\}\big) = \min\big\{\delta(\mathtt{a},\mathtt{d}), \delta(\mathtt{a},\mathtt{e})\big\} = \delta(\mathtt{a},\mathtt{b})$, and similarly, $d\big(\{\mathtt{a}\}, \{\mathtt{g},\mathtt{h}\}\big) = \delta(\mathtt{a},\mathtt{b})$.

3.2. Execution strategy

Before we discuss the execution strategy of PTIDES, one more definition needs to be introduced. For actor α, let $\mathbb{E}_\alpha$ be the set of equivalence classes containing its input ports, i.e., $\mathbb{E}_\alpha = \mathbb{E} \cap 2^{I_\alpha}$. Let $\mathcal{E}_\alpha$ be an arbitrary equivalence class that ranges over $\mathbb{E}_\alpha$. A *dependency cut* (or simply a *cut*) for $\mathcal{E}_\alpha$, denoted by $\mathbb{C}[\![\mathcal{E}_\alpha]\!] \subseteq \mathbb{E}$, is a minimal set of equivalence classes that satisfies the following.

> *For any $p \in \mathcal{E}_\alpha$ and any path ρ to p, there exist $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$, $p' \in \mathcal{E}'$ and ρ' being a path from p' to p, such that either ρ is a subpath of ρ' or ρ' is a subpath of ρ. (A subpath is a sequence of consecutive ports in a path. It is also called a substring in the literature.)*

Intuitively, a cut for $\mathcal{E}_\alpha$ is a "complete" set of equivalence classes on which $\mathcal{E}_\alpha$ depends. Completeness in this case means that for each port in $\mathcal{E}_\alpha$, all ports it depends on will be accounted for in $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$, either directly by being included or indirectly by having either upstream or downstream ports included.

Note that for $\mathcal{E}_\alpha$, the choice of $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ is not unique. For example, in Figure 2, $\big\{\{\mathtt{a}\}\big\}$ and $\big\{\{\mathtt{g},\mathtt{h}\}\big\}$ are both cuts for equivalence class $\{\mathtt{g},\mathtt{h}\}$. $\big\{\{\mathtt{d},\mathtt{e}\}\big\}$ is not, because if we pick port h and path $[\mathtt{c},\mathtt{h}]$, then no port in $\big\{\{\mathtt{d},\mathtt{e}\}\big\}$ is on this path or on another path that backward-extends this path. $\big\{\{\mathtt{d},\mathtt{e}\},\{\mathtt{g},\mathtt{h}\}\big\}$ is not a cut either, because it is not minimal. Minimality is a requirement for the efficiency of the PTIDES execution strategy, but not for its correctness. Redundant equivalence classes lead to unnecessary tests for events at the ports in those equivalence classes.

During execution, an actor α decides whether to process its input events by evaluating its *time-advance function* $\mathcal{T}_\alpha : \mathbb{E}_\alpha \times R^+ \times R^+ \to boolean$. The first argument is an equivalence class containing ports that belong to α, the second argument is a model time, and the third argument is a physical time in a certain time unit. A true value for $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ means that at physical time equal to or greater than τ, actor α will not receive any event e at a port in $\mathcal{E}_\alpha$ with $tag(e) < t$. Obviously, for any $t' < t$, $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau) \Rightarrow \mathcal{T}_\alpha(\mathcal{E}_\alpha, t', \tau)$, and for any $\tau' > \tau$, $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau) \Rightarrow \mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau')$.

The general strategy to evaluate $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ for model time t and physical time τ can be summarized as follows:

> *Let $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ be an arbitrarily chosen cut for $\mathcal{E}_\alpha$. $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ is true if for any $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$, α has received all the events at the ports in $\mathcal{E}_\alpha$ that depend on events at the ports in $\mathcal{E}'$ with time tags smaller than $t - d(\mathcal{E}', \mathcal{E}_\alpha)$.*

Various implementations can be devised from this general strategy. These implementations differ in 1) how to choose $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ and 2) how α decides whether it has received all the events at the ports in $\mathcal{E}_\alpha$ that depend on events at the ports in $\mathcal{E}'$ with time tags smaller than $t - d(\mathcal{E}', \mathcal{E}_\alpha)$. Two implementations will be discussed in Section 3.3.

If $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ equals true, then at physical time equal to or greater than τ, actor α will not receive events at the ports in $\mathcal{E}_\alpha$ with timestamps smaller than t. Therefore, α can process the event at any port in $\mathcal{E}_\alpha$ whose time tag is smaller than or equal to t and is the smallest among the time tags of the available events at the ports in $\mathcal{E}_\alpha$.

3.3. Implementations

In Section 3.2 we have discussed a strategy for actor α to decide whether it can process input events at the ports in $\mathcal{E}_\alpha$ with time tags smaller than or equal to model time t. Here we sketch two feasible implementations for evaluating $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$, where τ is the physical time at which the evaluation is performed. We will start with a particular implementation that turns out to be similar to the strategy proposed by Chandy and Misra [3]. We will then improve it to achieve better concurrency.

1. Recall that a decision about the choice of $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ needs to be made when developing an implementation. In this particular implementation, we shall always choose $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ to be $\{\mathcal{E}_\alpha\}$. ($\{\mathcal{E}_\alpha\}$ trivially satisfies the conditions for a cut.) Take Figure 2 as an example. The dependency cut chosen for equivalence class $\mathcal{E}_\alpha = \{\texttt{g}, \texttt{h}\}$ is $\{\mathcal{E}_\alpha\} = \{\{\texttt{g}, \texttt{h}\}\}$.

   In this case, the equivalence class $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$ to be tested is $\mathcal{E}_\alpha$ itself. Since $d(\mathcal{E}_\alpha, \mathcal{E}_\alpha) = 0$, the expression $t - d(\mathcal{E}_\alpha, \mathcal{E}_\alpha)$ is equal to t. Therefore, the strategy is equivalent to testing whether actor α has received all events at the ports in $\mathcal{E}_\alpha$ with time tags smaller than t.

   This implementation resembles the strategy by Chandy and Misra. The difference is that this one requires α to wait for events to be available at all the ports in $\mathcal{E}_\alpha \subseteq I_\alpha$, while Chandy and Misra's requires it to wait for events to be available at all the ports in I_α. (Our previous assumption that an actor's output events always depend on the most recent input events at its input ports actually equates $\mathcal{E}_\alpha$ with I_α, and makes this implementation equivalent to the strategy by Chandy and Misra.)

2. An improvement on the above implementation is inspired by time-synchronized distributed platforms that are made available by recent distributed clock synchronization protocols. In particular, computers in our experimental environment are equipped with clocks synchronized with the IEEE 1588 protocol [6]. The synchronization error of those clocks is bounded, and the network delay is also bounded.

   Each computer in our system receives external inputs from other computers or the sensor actors running on it. (Actors that receive data from the disk can also be considered as sensor actors.) Since the inputs from sensor actors can be regarded as inputs from computers in the external environment, without loss of generality, the following discussion focuses on computers that receive inputs solely from other computers.

   This implementation has two prerequisites. The first one is that the cut $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ for $\mathcal{E}_\alpha$ is chosen as the boundary between the computer that α runs on and other computers. More formally, the equivalence classes in $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ consist of input ports that belong to actors on the same computer with α, and are connected to output ports that belong to actors on other computers. For example, in Figure 1, the input ports of the Image Processor on the central computer comprise an equivalence class of input ports that are connected to the output ports belonging to the actors on other computers (in this case, the cameras). This prerequisite can be easily met by creating additional actors that receive inputs from other computers and relay those inputs to the original actors on the same computer. In this way, only the additional actors receive inputs from other computers, so the cut defined with the equivalence classes containing their input ports satisfies the prerequisite. In practice, these additional actors need not be actually created in the model.

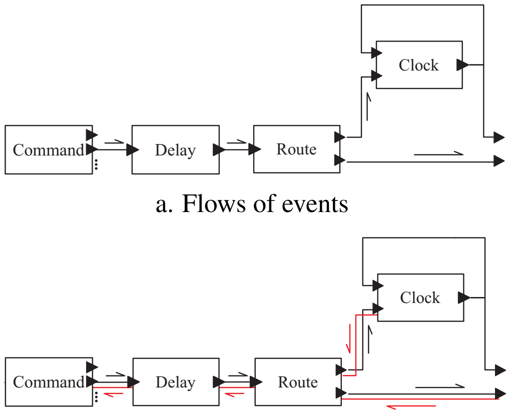
   The second prerequisite is that for any event e sent from another computer to the computer that α runs on, $tag(e)$ is always greater than or equal to the physical time at which e is output. With this, we can use the sender's current physical time as a lower bound of the time tags of all future events that it sends out. In our approach, this prerequisite is met by analyzing the maximum real-time delay between actors' input ports and output ports, and making sure that the events between computers be sent before the physical time reaches the time tags of those events. One can think of this prerequisite as a hard real-time constraint about deadlines that the system must meet. (Refer to [5] for a discussion on a precise timed (PRET) machine that provides underlying support for this real-time requirement.)

   Based on the above facts and prerequisites, $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$, where t is a model time that actor α intends to advance to in order to process input events, and τ is the current physical time measured on α's computer, can be evaluated as follows:

   > *$\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ is true if for any $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$ and any actor β whose output ports are connected to the ports in $\mathcal{E}'$, $\tau \geq t - d(\mathcal{E}', \mathcal{E}_\alpha) + \delta_{\beta,\alpha}$, where $\delta_{\beta,\alpha}$ is the maximum delay of the network connection from β to α.*

   At physical time τ, due to the bounded network delay, the ports in $\mathcal{E}'$ have received all events that β outputs before physical time $\tau - \delta_{\beta,\alpha}$. (Here, β must be on another computer due to the first prerequisite about the cut on the computer boundary.) The second prerequisite relates this physical time to model time, so the ports in $\mathcal{E}'$ have received all events from β with time tags smaller than $\tau - \delta_{\beta,\alpha}$. If $\tau - \delta_{\beta,\alpha} \geq t - d(\mathcal{E}', \mathcal{E}_\alpha)$, or equivalently $\tau \geq t - d(\mathcal{E}', \mathcal{E}_\alpha) + \delta_{\beta,\alpha}$, then the ports in $\mathcal{E}'$ have received all events from β with time tags less than $t - d(\mathcal{E}', \mathcal{E}_\alpha)$. If this is true for any equivalence class $\mathcal{E}'$ and any actor β that outputs events to the ports in $\mathcal{E}'$, then we can apply the execution strategy described in Section 3.2 to determine $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ to be true. We can thus allow α to process the event at any port in $\mathcal{E}$ with the smallest time tag that is also less than t.

   The cost of evaluating $\mathcal{T}_\alpha(\mathcal{E}_\alpha, t, \tau)$ is low because all $d(\mathcal{E}', \mathcal{E}_\alpha)$ and $\delta_{\beta,\alpha}$ are known statically. Actor α only



a. Flows of events

b. Flows of events and backward flows of hardware errors

Figure 3. Actor communication

needs to compare its physical time τ with a set of expressions in the form of $t - d(\mathcal{E}', \mathcal{E}_\alpha) + \delta_{\beta,\alpha}$, whose values can be easily computed.

If this implementation is used to execute the model in Figure 1, the distributed cameras and the central computer need to be equipped with synchronized clocks. The input ports of the Image Processor constitute an equivalence class $\mathcal{E}_\alpha$, where α refers to the Image Processor actor itself. The dependency cut $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ chosen by this implementation is $\{\mathcal{E}_\alpha\}$, which is the set of the equivalence classes on the computer boundary. Therefore, for any $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$, $d(\mathcal{E}', \mathcal{E}_\alpha)$ equals 0. The set of expressions are $\{t - \delta_{Dev_i,IP} | 1 \leq i \leq n\}$, where t is the model time of the earliest image that the Image Processor needs to process, and $\delta_{Dev_i,IP}$ is the maximum delay of the network connection from the i-th Device actor to the Image Processor.

3.4. Execution with errors

According to the execution strategy in Section 3.2, for α to decide whether it can process event e with $tag(e) \leq t$ at an input port in $\mathcal{E}_\alpha$, $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ is chosen according to the implementation, and for any $\mathcal{E}' \in \mathbb{C}[\![\mathcal{E}_\alpha]\!]$, α must have received all the events at the ports in $\mathcal{E}_\alpha$ that depend on events at the ports in $\mathcal{E}'$ with time tags smaller than $t - d(\mathcal{E}', \mathcal{E}_\alpha)$. We prefer $d(\mathcal{E}', \mathcal{E}_\alpha)$ for each $\mathcal{E}'$ to be a large positive number, because this allows α to process events earlier. However, when hardware errors are considered, $d(\mathcal{E}', \mathcal{E}_\alpha)$ may become 0.

Figure 3 shows the communication between the Command actor and any of the n Clock actors in our application. The Clock repeatedly schedules its future activity according to the current frequency. When it receives a "change frequency" command, it cancels the imminent schedule and makes a new schedule. If the new frequency is too small, the clock refuses to reschedule because the real-time delay on event processing does not allow it to actuate the downstream Device actor on time. In this case, it sends back an *error event* to its predecessor (the Route actor). Otherwise, it sends back a *success event* instead.

An error event carries the same time tag as that of the command that caused the error. Because it may receive an error event, the Route actor cannot perform any further computation after it sends a command to the Clock actor until it receives a success event or an error event (unless it can determine that the potential error does not affect its computation result or internal state, which is not true for the general-purpose Route actor). Furthermore, on receiving an error event, the Route actor is not capable of handling it. It has to send it back to the Delay actor, which in turn sends the error back to the Command actor for handling. As a result, like the Route actor, the Command and the Delay cannot perform further computation until they receive either a success event or an error event.

This lack of concurrency is because the minimum delay δ between these actors' command input ports and error event output ports is 0. The relevant dependency between the equivalence classes containing these ports, computed with δ, is also 0. We will revisit this problem in Section 5, and provide a solution with backtracking.

4. Backtracking Based on Incremental Checkpointing and State Recovery

In this section, we discuss a backtracking mechanism [8] that can be incorporated into PTIDES as an extension. It allows general-purpose actors to perform computation without always waiting for success events or error events. If errors occur during the hardware operations, the state of the actors can be recovered with the backtracking mechanism.

In our model execution framework (Ptolemy II [2]), actors are either atomic ones or composite ones. Atomic actors are defined as classes in the Java programming language. Composite actors are hierarchical composition of actors (atomic or composite).

At the beginning of an execution, atomic actors are instantiated from their classes. State may be stored in the instances' fields. We call the atomic actors that maintain state *stateful actors*.

We have implemented a program transformer to enhance stateful actors' source code with incremental checkpoint functionality, which the actors can use to backtrack to their previous state. The transformation preserves the actors' behavior perceivable from the execution result. For most general-purpose actors, this transformation is fully automatic and requires no human intervention. For actors that store state in special forms such as files in a network file system, the actor designers can customize the incremental checkpointing with extra manually written functions. The incremental checkpointing functionality incurs low run-time overhead, which will be analyzed at the end of this section.

The discussion here focuses on the Java programming language and the state stored in non-static fields. However, similar techniques can be applied to other programming languages, and our approach can be easily extended to support state stored elsewhere, such as that in local variables, static fields, and files on disks.

4.1. Recording state changes

The program transformer generates extra code in the actors' classes to keep track of state changes. This code can be enabled or disabled at run time. When it is enabled, a handler is invoked in place of any state change. It stores the old state in a record object in memory before the change is made. The record objects can be used to restore the state later.

Two sources of state changes are captured by the handlers: assignments and routines in external libraries. Assignments can be detected by a search in the actor classes' abstract syntax trees (AST's). These AST's are obtained from a Java parser, and a type resolver annotates them with type information. The transformer replaces each assignment with an invocation of a handler. For calls to routines in external libraries that potentially change the state, such as `put()` in class `Hashtable` and `arraycopy()` in `System`, substitution methods are generated to record the state before performing the change. For example, for the `put()` method in `Hashtable`, a new `Hashtable` class is generated, which provides a modified `put()` method. This `put()` method stores the old value associated with the affected key in a record object, and then performs the same change as the original `put()` method. The transformer then changes the type of the instance to the new `Hashtable` class, so that the modified `put()` method is called instead of the one in Java's standard library. This transformation, as well as the generation of the modified `Hashtable` class, is fully automatic.

Table 1 shows an example of transforming an integer assignment. `$ASSIGN$s()` is the handler generated to substitute the assignment to field `s`. It allocates a record object in memory, and stores the current value of field `s` in it. The record objects for `s` are linked and ordered by their creation time. (The code for recording the value in a newly created record object is omitted because it add unnecessary complication to our discussion.)

Besides primitive data, the state of objects and arrays can also be recorded. For those types, references are recorded instead of the content. An observation about the handlers is that the time taken to record an old value is always constant, whether the field assigned to is primitive or not [8]. This time is spent on allocating the record object and storing the the old primitive value or the old reference of the object or array in it.

```
int s;
void f(int i) {
  s = i;
}
```

a. Before Transformation

```
int s;
void f(int i) {
  // s = i;
  $ASSIGN$s(i);
}
int $ASSIGN$s(int newS) {
  ... // Record the current value of field s
  return s = newS;
}
```

b. After Transformation

Table 1. Transformation of integer assignment

4.2. Checkpoint management

A *checkpoint* is an execution point of an actor with state associated with it. We assume that no state is shared between actors. This is a fair assumption for actor-oriented models, in which actors exchange information by means of events sent or received via output ports and input ports, respectively. We provide a small yet sufficient set of methods for the actors to manage their checkpoints. For most general-purpose actors, this checkpoint management is also automatic.

The `checkpoint()` method in the library that we provide can be invoked to create a checkpoint. It returns a long value as a unique *checkpoint handle*. After checkpoint creation, the assignment handlers are enabled to record state changes. Later, the actor can backtrack to the checkpoint by invoking `rollback()` with the checkpoint handle as parameter. An actor can maintain multiple checkpoints at the same time, and it can roll back to any of those checkpoints. When an actor decides that it will not backtrack to a checkpoint, it invokes the `discard()` method with the checkpoint handle to free the memory consumed by the record objects. References to those record objects will be lost, and the Java garbage collector will later collect the memory for reuse.

Our backtracking mechanism is implemented specially for applications in which run-time performance is important. The `checkpoint()` method incurs a small constant cost in time and in memory. It marks the current execution point with a unique long integer number without actually storing any state. The state is stored incrementally with assignments. There is a constant run-time cost for each assignment. Assuming that assignments are performed uniformly during the execution, the handlers slow down the execution at a fixed rate that can be statically analyzed to extract real-time properties. `rollback()` and `discard()` have $O(cv)$ complexity, where c is the maximum number of checkpoints that coexist at any time, and v is the number of assignable memory locations (e.g., integer fields, object references, array elements, etc).

5. Enhanced Camera Application

We now apply the backtracking mechanism to the image composition application. The backtracking mechanism provides fault tolerance at the software level without requiring a redesign of most actors in our actor library.

We will consider the Delay, Route, Merge, Queue and Image Processor actors in this application to be general-purpose. They have no knowledge about the meaning of error events or how to handle them. We will not discuss state recovery of other actors that interact with hardware or the human operator, because that depends on the specific operations provided by the API. For example, the Command actor in our application was considered a sensor. It may keep a history of previous commands. Checkpoints of this actor correspond to the indices in the history. To recover state with a checkpoint, the Command actor may simply restore the history to the specified index, and delete the subsequent commands. Another example is the Display actor, considered as an actuator. It can not backtrack because the display of erroneous data cannot revert. One way to deal with this is to allocate a large enough buffer to store composed images, which are shown on the display device only when the actor is sure that the data used to compose them are valid.

In the following discussion, we will focus on general-purpose actors. In general, the computation on those actors depends on the successful completion of hardware commands in the downstream actuators. On receiving an error event, the default behavior of those actors is to recover their state to the point before the event pertaining to the problematic command was processed, and to relay the error event back to their predecessors.

5.1. Processing error events

We will use the Image Processor actor as an example to discuss the processing of error events in general-purpose actors.

The Image Processor actor composes the images acquired at the same time from the n cameras. It is a stateful actor because historical data about past images (up to a certain time) may be used in the current composition. Due to temporary loss of network connections, camera malfunction, or failure to focus, the Image Processor may not receive all the n images on occasions, and some of the received images may be corrupted. We assume there is a way for the Image Processor to detect lost or corrupted images. A simple example is that if it does not receive data of a partially transmitted image for a threshold, it considers the connection lost, and the partial image corrupted.

A question to ask is when the Image Processor can start to compose the images. If the composition starts at the time when the Image Processor begins to receive image data from any camera, and if during the composition some images are found to be corrupted, then the changes made on the historical data are invalidated and the historical data cannot be used any more. However, if the composition starts when all the n images are either received intact or found corrupted, then it may be too late the Display actor to produce real-time display, because image composition takes a considerable amount of time.

This problem is similar to the one we have seen in Section 3.4. We let Image Processor be actor α here. Let e_1 be an event at $i_1 \in I_\alpha$ that represents the start of transferring an image, which later becomes corrupted. Let e_2 be an error event at $i_2 \in I_\alpha$ that represents the detection of the corruption. $tag(e_1) = tag(e_2)$ because e_2 invalidates any computation performed after e_1. i_1 and i_2 are in the same equivalence class because they both have influence on α's output signal (the set of composed images). Let this equivalence class be $\mathcal{E}_\alpha$. Since e_2 originates from the Device actors and does not depend any other event, the only path to i_2 is $[i_2]$. Therefore, the only $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ that α can choose is $\{\mathcal{E}_\alpha\}$. As we have seen, this choice of $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ makes the PTIDES execution strategy equivalent to Chandy and Misra's conservative strategy. α has to pessimistically wait for e_2 (or a success event at port i_2 instead) in order to process any event with a time tag greater than $tag(e_1)$.

With backtracking, the Image Processor can perform composition based on partially received data after it creates a checkpoint. Changes can be made on the historical data. The old values of the changed data are automatically recorded without extra programming. In the rare case where image corruption is detected, the Image Processor invokes `rollback()` to recover its previous state, and then restarts the composition with fewer images. In the majority of cases, image transfer and composition are performed in parallel and are both successful. Even in the rare case where composition has to be restarted, the time to finish the composition is similar to the time taken by traditional conservative strategies, not considering the part of composition that can be reused after state recovery.

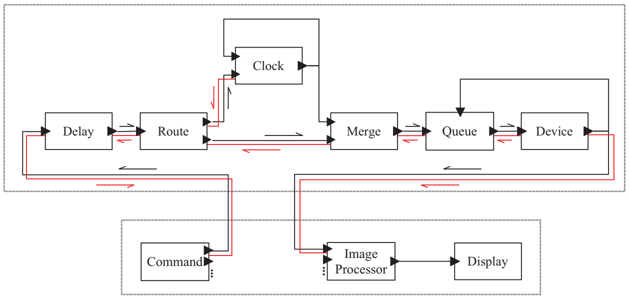


Figure 4. The camera application enhanced with error channels

5.2. Error channels and cancellation messages

Figure 4 shows the model enhanced with *error channels*. Device and Clock are the actors that generate error events. The errors propagate via error channels to other actors, which may further relay them. Errors stop propagating until they are properly handled. (For loss of connections between the n Device actors and the Image Processor, we shall consider that the Image Processor "receives" an error event from a Device if it has not received data from that Device for a certain timeout.)

In the static analysis of relevant dependency before execution, the error channels are deliberately ignored, and so are the error events that travel via those channels. Hence, the static analysis for relevant dependency is not affected by hardware errors. The Image Processor can choose a $\mathbb{C}[\![\mathcal{E}_\alpha]\!]$ different from $\{\mathcal{E}_\alpha\}$. For example, it can choose the set containing n equivalence classes, each of which consists of an input port of a distinct Delay actor. Using the second implementation in Section 3.3, the Image Processor can decide whether it can process input events by comparing the physical time of the central computer to the expressions involving the relevant dependency between those equivalence classes and $\{\mathcal{E}_\alpha\}$.

At run time, hardware errors are automatically handled by general-purpose actors. An actor creates a checkpoint every time it starts to process an event. To continue with the example of Image Processor, every time it starts to receive images from the cameras, it creates a checkpoint and proceeds with the image composition without waiting. This checkpoint will be useful when it receives an error event (e.g., a one corresponding to the loss of a network connection). At that time, it recovers its state by backtracking to the checkpoint. Note that the checkpoint management functionality itself is not subject to backtracking, so it can maintain the list of events that have been received after the checkpoint creation. Those events may need to be re-processed after state recovery.

As a result of state recovery, an actor may generate *cancellation messages* to other actors. This is because some of the events that it previously sent to those actors become invalid. Like error events, the cancellation messages are sent via the error channels, and they are not considered in the static analysis.

Cancellation messages in distributed systems have been studied by previous research. For example, a cancellation mechanism for Time Warp is described in [4]. Cascading failure, also known as domino effect [13], must be avoided. This problem is due to the fact that the cost of checkpoint creation in the prior work is high, so it is not practical to create a checkpoint every time an event is to be processed. As a consequence, to cancel a previously processed event e, an actor may backtrack to a model time earlier than $tag(e)$. It may then cancel its output e' earlier than e. For the actor that received e' to cancel it, some outputs of that actor earlier than $tag(e')$ may also need to be canceled for the same reason. If this goes on, in the extreme case, the whole distributed system backtracks to very far in its past, or even to its initial state.

In our backtracking mechanism, checkpoint creation is an inexpensive operation that an actor can always perform before it processes an event. To cancel the processing of an event, the actor backtracks to the checkpoint created right before that event was processed. This assures that the domino effect does not happen.

6. Assessment and Related Work

Our backtracking mechanism has low performance overhead and small memory footprint. It is best suited for applications with real-time requirements. If assignments are distributed uniformly in the execution, the extra state recording slows down the applications at a fixed rate. This makes it easy to infer real-time properties based on the real-time properties of the non-backtracking version of the applications. Since error events are rare, the rollback operation does not cause significant performance penalty. Nevertheless, for time-critical applications that require strict conformance to real-time constraints, the time consumed by rollback still needs to be carefully considered. Fortunately, the cost can be analyzed with the maximum number of coexisting checkpoints (1 in many actors) and the number of assignable locations.

Compared to the traditional conservative strategy [3, 12], PTIDES takes advantage of the provable minimum delay to process events more eagerly, making actors more responsive to incoming events. Compared to optimistic strategies [9, 4], PTIDES requires much less backtracking, which is needed only when hardware errors occur. Optimistic strategies require backtracking to deal with time faults. The extra cost and the unanalyzable nature of time faults sometimes outbid the gain of optimistic event processing.

Various techniques exist for software fault tolerance [7]. We find that some of them do not satisfy the requirement of real-time applications, while the others require too much human intervention and are not flexible enough. For example, serialization at checkpoint creation makes it necessary to pause the applications for a considerable amount of time. It cannot be used when real-time performance is a critical concern. The incremental checkpointing technique described in [10] requires the programmers (in our case, designers of atomic actors) to provide auxiliary methods, which are an extra burden and a source of bugs. It is also hard to maintain these methods as the code evolves. Our backtracking mechanism does not have these limitations. The programmers only maintain the original copy of the code, which has no backtracking functionality. When the code is updated, transformation is applied with a tool to generate a new backtracking version. This process can be incorporated into an unattended build system. No human intervention is needed for most general-purpose actors.

7. Conclusion

We develop PTIDES for the execution of real-time distributed discrete-event embedded systems. A lightweight backtracking mechanism based on program transformation is incorporated into PTIDES as an extension. It provides better concurrency for event processing, because the actors need not always wait for success events or error events from the hardware. In the rare case where the hardware cannot complete the operations successfully and error events are produced, the actors automatically backtrack to their previous state. Our backtracking mechanism has low overhead in execution time and memory resource. It smoothly slows down the execution, making it possible to prove real-time properties of the applications.

Acknowledgment

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, National Instruments, and Toyota.

References

[1] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, New York, 1992.

[2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Ptolemy II - heterogeneous concurrent modeling and design in Java. Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley, Jan 2007.

[3] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transaction on Software Engineering*, 5(5), 1979.

[4] M. Chetlur and P. A. Wilsey. Causality representation and cancellation mechanism in time warp simulations. In *PADS '01: Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 165–172, Washington, DC, USA, 2001. IEEE Computer Society.

[5] S. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, Nov 2006.

[6] J. Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, pages 98–105, Nov 2002.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[8] T. H. Feng and E. A. Lee. Incremental checkpointing with application to distributed discrete event simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1004–1011. Winter Simulation Conference, 2006.

[9] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[10] J. L. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70, New York, NY, USA, 2000. IEEE.

[11] Z. Manna and A. Pnueli. Verifying hybrid systems. In *Hybrid Systems*, pages 4–35, London, UK, 1993. Springer-Verlag.

[12] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[13] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press.

[14] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07)*, pages 259–268, Bellevue, WA, USA, Apr 2007.