Distributed Simulation of Multi-Agent Hybrid Systems

Yerang Hur and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
200 S 33rd St., Philadelphia, USA
{yehur,lee}@cis.upenn.edu

Abstract

Systems such as coordinating robot systems, automobiles, aircrafts, and chemical process control systems can be modeled as interacting hybrid systems, where hybrid systems are finite state machines with continuous dynamics. The language CHARON *and its simulator have been developed to model and analyze interacting hybrid systems as communicating agents.*

Simulations are widely used for the analyses of hybrid systems. The simulation of a complex system is, however, usually very slow. This paper proposes four algorithms for distributed simulations of hybrid systems. The idea behind distributed simulations is to achieve a speedup by utilizing multiple computing resources. The agents of a modeled system are distributed over multiple processors to simulate the agents more efficiently. Since the state of the agent is affected by the input from other agents, they synchronize to update their local states. The challenge here is how to reduce the agent synchronization overhead.

We present two approaches for resolving the problem: conservative and optimistic approaches. For the optimistic approach, we present three different algorithms for distributed simulations of hybrid systems, and compare them.

1 Introduction

Coordinating robot control systems, electro-mechanical systems, or chemical process control systems have a common feature that they control continuous dynamics with discrete mode changes. The hybrid system community composed of researchers from control engineers and computer scientists has evolved to support modeling, simulation, verification, and code generation for such dynamical systems. Traditional engineering tools, provide little support for developing complex hybrid systems with modes changing dynamics in response to conditions or discrete events.

There have been studies concerning the language support for modeling hybrid systems to help people build complex hybrid systems for years. With an introduction of an object-oriented design paradigm, researchers developed object-oriented modeling languages for hybrid systems. Among those are Omola [15], Dymola [8], SHIFT [7], and Modelica [9]. Those languages, however, do not provide formal semantics necessary for reasoning about the model of hybrid systems.

Compared to the previous languages for the modeling of hybrid systems, the motivation of developing CHARON was to provide a formal way of verifying hybrid systems and to generate the executable code from models described in a language for modeling both the architecture and the behavior of hybrid systems modularly. Additionally, CHARON supports exception handlings through group transitions and default control points and provides interfaces between models and external packages described in Java [3, 4]. CHARON has been used to model and study a multi-robot coordination [2], the Simplex Architecture-based inverted pendulum controller [11], and automotive vehicle systems [1].

Simulations have been widely-used for analyzing hybrid systems. Simulations of complex hybrid systems are, however, usually very slow. This paper describes four algorithms that we have developed for the distributed simulation of hybrid systems. We simulate hybrid systems in a distributed fashion to achieve a speedup by utilizing multiple computing resources. The agents of the modeled system are distributed over multiple processors to simulate the model more efficiently.

Based on how agents synchronize, distributed simulations of multi-agent hybrid systems can be classified as $conservative$ and $optimistic$ simulations. If the local clock of an agent always advances and does not go backward, it is called a $conservative\ simulation$; otherwise, it is called an $optimistic\ simulation$. A conservative simulation is designed so that the local clock lc of an agent only advances or stops, and thus, there are no rollbacks [6]. In optimistic simulations, the goal is to exploit the potential parallelism as much as possible letting the agents run at different speeds. If an event that occurred at time t_e gets recognized by the agent at t_r, where $t_r > t_e$, the simulator provides a rollback operation or performs a reverse computation [13, 5].

Distributed simulations have been used for simulating only discrete event systems so far, but to the best of our knowledge, there is no published algorithm for the distributed simulation of hybrid systems. Our methods to simulate hybrid systems in a distributed fashion exploit the inherent modularity of systems described in CHARON. By modularity, we mean two things. One is a behavioral modularity captured by modes and the other is an architectural modularity by agents. A way of using a mode-level modularity in single-agent hybrid systems was presented in [3]. In this paper we focus on distributing atomic agents to utilize an agent-level modularity. The challenge is how to reduce the synchronization overhead among distributed agents.

In the following sections, we introduce a language, called CHARON, for modeling interactive multi-agent hybrid systems and its toolset in Section 2. We describe how to simulate hybrid systems in Section 3, propose four distributed simulation algorithms of hybrid systems in Section 4, and discuss the proposed algorithms and future work in Section 5.

2 Modeling Hybrid Systems in CHARON

CHARON [3] is a language for modeling interacting hybrid systems based on the notions of agents and modes. The distinguished feature of CHARON is that it supports both architectural and behavioral descriptions of hybrid systems and provides concurrency and hierarchy in a modular way. It also has the scoping rule of variables partitioned into the set of analog variables and discrete variables. Agents describe the architecture of systems while modes do the behavior of systems. CHARON modes comprise submodes, transition relations between submodes, control points, differential and algebraic constraints, invariants, and variables [3]. The language provides compositional formal semantics required to reason about systems in a modular way [4]. We now briefly describe the main features of CHARON.

Architectural hierarchy. The architecture of systems is described with communicating *agents*. Those agents share information through shared variables or communication channels. Agents are either *atomic* or *composite*. CHARON also provides the operations of *parallel composition* for building a *composite* agent, *hiding* for encapsulating information of an agent. CHARON agents are reusable objects.

Behavioral hierarchy. An agent without any submodes inside is called *atomic agent*, which has a reference to an appropriate mode describing the behavior of the agent. In other words, the mode is a construct for the hierarchical description of the behavior. Transitions between submodes are enabled when a condition called *guard* becomes true. Also, the mode has well-defined control points: *entry* and *exit* points. We provide differential and algebraic constraints representing continuous dynamics and invariants forcing a continuous flow to satisfy a condition. The language also supports the instantiation of a mode for the reuse of mode definitions.

Continuous variables and discrete variables. The variables of CHARON are classified as *analog* and *discrete* variables. *Analog* variables are updated continuously while time is flowing. Conversely, discrete variables are modified only when the modes of an agent change. The values of discrete variables do not change in a time flow.

2.1 Example: TwoAgent

We present a simple two-agent example, *TwoAgent*, composed of the agent *a1* and the agent *a2*. The dynamics of *a1* is independent of that of *a2* but guards of agents require shared information to decide whether an agent changes modes or not.

```
macro TooLow 0.0
extern real Math.abs(real);
agent TwoAgent () {
   private analog real v1, v2;
   agent a1 = A(10.0, 0.0)
   [vIn, vOut := v2, v1];
   agent a2 = A(9.0, -1.0)
   [vIn, vOut := v1, v2];
}
agent A(real initValue, real c){
   read analog real vIn;
   write analog real vOut;
   mode top=ATop(initValue, c);
}
```

Figure 1. Agent TwoAgent

Figure 1 defines a composite agent *TwoAgent* and an atomic agent *A*. The two atomic agents *a1* and *a2* of the composite agent *TwoAgent* are the instances of the agent definition *A*. The agent *A* has a reference to its top-mode *top* that describes the behavior of *A*. The behavior of the agent *A* is represented in two submodes: *mode0* and *mode1*. Both submodes are instances of the mode *choppy* but parameters are different. We use *extern* to specify interfaces between the CHARON code and external packages. Subagents of a composite agent communicate through shared variables whose name are renamed. `[vIn, vOut := v2, v1]` and `[vIn, vOut := v1, v2]` are the examples of renaming. *vIn* of the agent *a1* and *vOut* of the agent *a2* are renamed to the new name *v2*. Thus, if *a2* updates *vOut* then it actually updates the shared variable *v2* and so, *vIn* of *a1* is also updated.

Figure 2 is the CHARON code describing the behavior of *A*. Our CHARON toolset provides an editor with GUI (Graphical User Interface) for the visual specification of agents and modes. Figure 3 shows how the behavior of A is specified visually.

The mode definition *ATop* comprises variables *vIn* and *vOut*, submodes *mode0* and *mode1*, and transitions. Transitions are represented with *trans* construct. In this example, we have three transitions: *initTrans*, *Mode0ToMode1*, and *Mode1ToMode0*. Action statements provided by *do* are for specifying actions taken when transitions occur. Basically, actions update discrete variables or assign new values to continuous variables.

Entry/exit points are for specifying interfaces between the source locations and the destination locations of transitions. The variable *vOut* is declared as *write*, which means global and writable in the mode while *read* means read-only. For localizing the scope of the variables, we use *private*. The invariant *invChoppy* expresses that if *vOut* is out of the specified range, the behavior of *choppy* is no longer valid. We specify the rate at which *vOut* is updated with the differential equation *dvOut*.

Figure 4 is the result of simulating the example. Each agent generates a signal like a choppy wave, where the

```
mode ATop(real iVal, real c){
   read analog real vIn;
   write analog real vOut;
   mode mode0 = choppy(2.0, -50.0, c);
   mode mode1 = choppy(2.0, 1.0, c);
   trans initTrans
     from default to mode0
     when true do {vOut = iVal}
   trans Mode0ToMode1
     from mode0 to mode1
     when vOut < 8.0
     and Math.abs(vOut-vIn) > 1.1 do {}
   trans Mode1ToMode0
     from mode1 to mode0
     when vOut > 12.0
     and Math.abs(vOut-vIn) > 1.0 do {}
}
mode choppy(real a, real b, real c){
   write analog real vOut;
   inv invChoppy {vOut >= TooLow}
   diff dvOut {d(vOut) == a*vOut + b + c}
}
```

Figure 2. Modes of TwoAgent

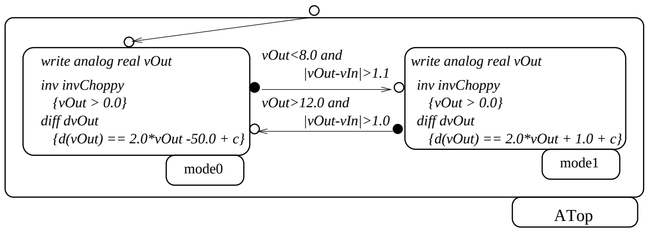


Figure 3. The Behavioral Hierarchy of the Agent A

wave of *a1* follows that of *a2*. In *a1*, the guards enabling the transitions $Mode0ToMode1$ and $Mode1ToMode0$ are $(v1 < 8.0\ and\ |v1 - v2| > 1.1)$ and $(v1 > 12.0\ and\ |v1-v2| > 1.0)$, respectively. Figure 4 shows that the mode transitions $Mode0ToMode1$ of *a1* occur at time 0.09, 0.50, 0.87, 1.21, and so on.

2.2 CHARON toolset

Figure 5 describes the class hierarchy of the TwoAgent example. We depict the classes and objects of the example [12] with a modified OMT (Object Modeling Technique) notation used by Gamma and his colleagues. The symbols of classes and objects are rectangular boxes and rounded boxes, respectively. The subclass relationship is represented with a line and a triangle. Dashed lines with filled arrowheads are for specifying instantiations. Figure 6 illustrates how CHARON Simulator Generator (CSG) generates the simulation code. We have two kinds of generators to produce a simulator for the example. One is *agentModeGen* that generates agents and modes and the other is *gSimGen*. *gSimGen* generates model-specific information the simulator core requires for simulating a given CHARON model. In other words, *simulator* comprises *simulator core* and the routines generated by *gSimGen*. Agents and modes generated by *agentModeGen* are Java files translated from the model described in CHARON. In essence we simulate the objects generated from a given CHARON model.

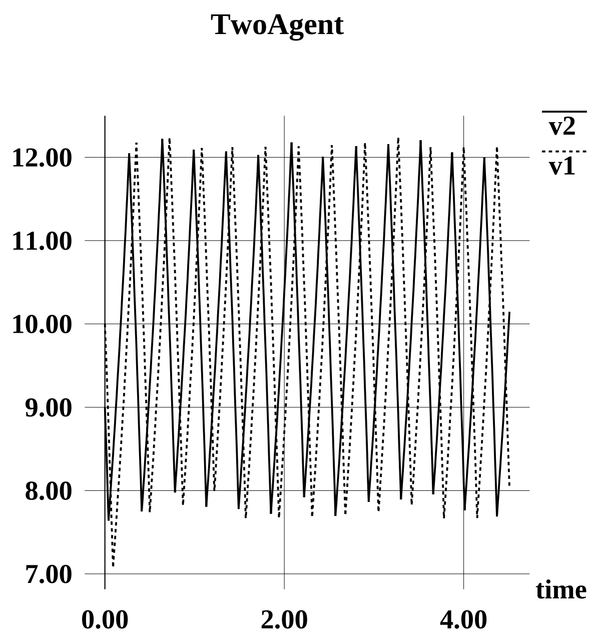


Figure 4. Simulation Result

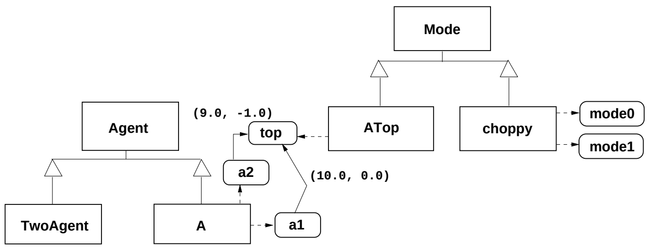
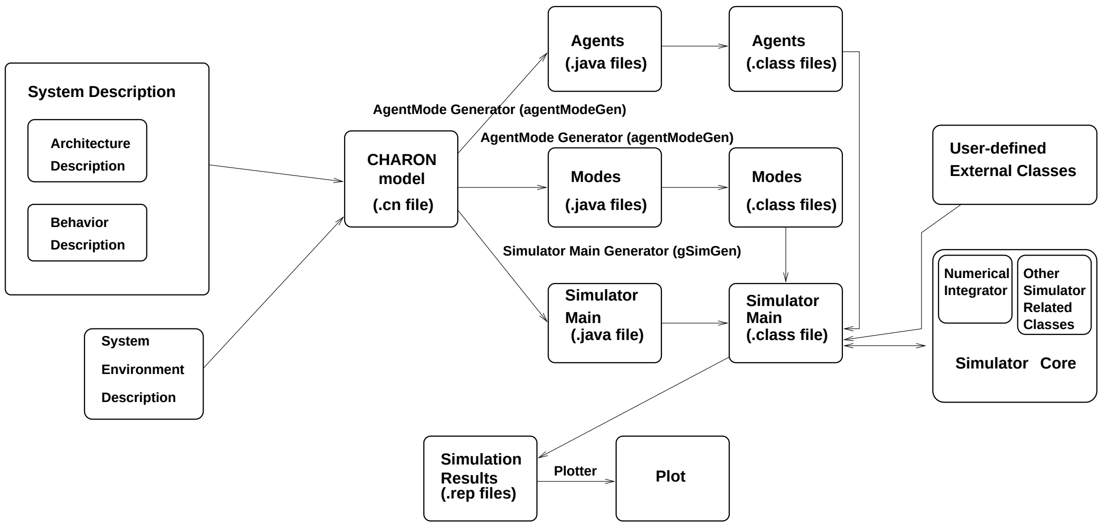


Figure 5. Classes and Objects of the TwoAgent Simulator

3 Simulation of CHARON Program

3.1 Global Simulation

In this section, we describe how the simulator simulates hybrid systems modeled in CHARON. The simulation of CHARON program consists of the initialization step, followed by a sequence of discrete update steps and continuous flow steps. After initializing agents, the simulator executes either a discrete update step or a continuous



System Description
Architecture Description
Behavior Description
System Environment Description
CHARON model (.cn file)
AgentMode Generator (agentModeGen)
AgentMode Generator (agentModeGen)
Simulator Main Generator (gSimGen)
Agents (.java files)
Agents (.class files)
Modes (.java files)
Modes (.class files)
Simulator Main (.java file)
Simulator Main (.class file)
User-defined External Classes
Numerical Integrator
Other Simulator Related Classes
Simulator Core
Simulation Results (.rep files)
Plotter
Plot

Figure 6. CHARON **Simulator Generation Process**

flow step. In other words, the simulation of CHARON program is *initialization_step* (*discrete_update_step* or *continuous_flow_step*)∗.

Initialization. The simulator initializes agents by providing initial values for variables and configuring active modes. With initialization done, the active modes of an agent are the modes included in the path from the top mode to a leaf mode. For example, in *TwoAgent*, *top* and *mode0* are the active modes and the initial values of *vOut* of *a1* and *a2* are 10.0 and 9.0, respectively after initializing the agents.

Discrete update. A discrete update step is a sequence of executing enabled transitions. The logical time of the simulator stops during transitions. The simulator picks an agent with enabled transitions. We say a transition is enabled if it is in an active mode and its associated guard is satisfied. After a transition is executed, the configuration of active modes is also updated. The step finishes if there is no more enabled transitions at the current time. In the example, the transition *Mode0ToMode1* will be enabled when *vOut* gets less than 8.0 and the difference between *vOut* and *vIn* greater than 1.1.

Continuous flow. A continuous flow step represents a time flow for a given amount of duration not violating constraints. The constraints of a mode consist of invariants, differential and algebraic constraints. In this step, all the active modes of agents are executed together. The constraints of active modes are, the conjunction of constraints in the set of active modes. The discrete update step may preempt the continuous flow step, if a transition is enabled. For example, if active modes are *top* and *mode0*, the variable *vOut* will get updated at the rate of $a * vOut + b + c$ as long as the invariant *invChoppy* is not violated. The transition *Mode0ToMode1*, however, may be executed when the guard of *trans Mode0ToMode1* becomes true.

3.2 Efficient Simulation

The most simulation overhead results from numerical integrations. For more efficient numerical integrations, various numerical integrators are combined with adaptive integration steps, e.g., Runge-Kutta method with adaptive integration steps [17]. Another approach is simulating the system on a distributed platform. Our methods proposed here exploit the inherent modularity of systems described in CHARON as well as simulate the systems in a distributed fashion. By modularity, we mean two things: a behavioral modularity and an architectural modularity. In fact, each agent of CHARON has different dynamics and thus does not have to run at the same speed. Likewise, each mode in the different level of hierarchy of the same agent may have different dynamics. The basic idea is that we simulate an agent or a mode of slow dynamics using a larger integration step. Therefore, we can reduce the computation overhead from integrations. A way to exploit a mode-level modularity in a single-agent was presented in [3]. In the next section we focus on distributing atomic agents to utilize an agent-level modularity.

4 Distributed Simulation

We distribute agents of the modeled system over a distributed system to simulate the model more efficiently. As agents share information, they need to synchronize to update states. According to the way of synchronization, distributed simulations can be classified as two categories. If the local clock of an agent always advances and does not go backward, it is called *conservative simulation*. Otherwise, it is called *optimistic simulation*.

4.1 Conservative Simulation

In this approach, we decompose the functionality of a global simulation into sub-functional blocks and the simulator allows agents to execute the next block only when all the agents complete the current block.

Figure 7 and Figure 8 describe the algorithm Conservative Simulation (CS). We decompose a simulation process into seven functional units: block0, ..., block6. The end of each block works as a barrier an agent cannot proceed until all the other agents finish the current block. Agents broadcast *completion* messages to the others at the end of each block.

In block0, we initialize the agent. The *analog* variables are updated in block1 followed by block2, where the shared *analog* variables are synchronized. In block3, guards and invariants are checked. Block4 is for executing enabled transitions. The variables updated in the transitions are synchronized in block5. Block6 is executed only when there is no enabled transition even though an invariant is violated. In such a case, the agent broadcasts *stop signal* to the others and the simulator reports the latest synchronized state.

Although CS is for the distributed simulation of hybrid systems, the overhead resulting from communications offset a potential performance gain from distributing computations. Thus, this technique is effective only in simulating hybrid systems with extensive computations. In Section 4.2 we propose three optimistic simulation algorithms to solve the problem.

```
begin block0
   state = initialize();
end
while(true){
   begin block1
     //update analog variables
     //for a given time interval ti
     //and return the new state
     //and the new lc.
     (state, lc)
     := continuousFlow(state, lc, ti);
   end
   begin block2
     // exchange shared analog variables.
     state
     := synchronizeSharedVars(sharedVars);
   end
   begin block3
     if evaluateGuards(state, guards)==true
     then event := transitionEnabled
     elseif checkInvariants(state,
       invariants)==true
     then event := invariantViolated
   end
```

Figure 7. Conservative Simulation (CS): block0 ... block3

```
   begin block4
     if event == transitionEnabled then
       state
       := discreteUpdate(state, lc);
   end
   begin block5
     //exchange shared variables updated
     //by discreteUpdate.
     state
     := synchronizeSharedVars(sharedVars)
   end
   begin block6
     if event == invariantViolated
     then broadcastStopSignal; stop
   end
}
```

Figure 8. Conservative Simulation (CS): block4 ... block6

4.2 Optimistic Simulation

Our optimistic simulation algorithms are phase-driven. Each phase is a sequence of consecutive stages that are setup stage, computation stage, and synchronization stage. After initializing the state, setup stage, computation stage, and synchronization stage are executed in the order as shown in Figure 9. The motivation of using the notion of phase is that we confine optimism within each phase.

In each of the Phase-driven Optimistic Simulation (POS), the operations performed during the stages are different. We describe the details of each stage of the three different POS approaches: Phase-driven Optimistic Simulation with a Reflection-based Prediction (POSRP), Phase-driven Optimistic Simulation with an Adaptive Phase Length (POSAPL), and Phase-driven Optimistic Simulation with an Estimator-based Prediction (POSEP).

The salient aspects of our algorithms are first, to reduce the communication overhead, we let agents synchronize just before the new value of a shared variable is necessary, instead of communicating every update round. Second, to reduce the computation overhead due to numerical integrations, we simulate an agent with its approximated polynomial dynamics and resolve the possible misses of events with rollback operations. This allows each agent to run computations without integrating shared variables controlled by other agents. Our approach is $optimistic$ in the sense that each agent goes forward even though there is no guarantee that their clocks do not have to go backward.

Optimistic distributed simulations have been used for discrete event systems since the $Time\ Warp$ distributed simulation algorithm was proposed [13]. Other methods in the category are the bounded-lag optimistic simulation [14], and a risk-free simulation [18]. Those methods are, however, only for simulating discrete event systems. To illustrate our algorithms, we use the two-agent system described in Section 2.

```
initialization;
while(!stop){
   executeSetupStage;
   executeComputationStage;
   executeSynchronizationStage;
}
```

Figure 9. Phase-driven Optimistic Simulation (POS)

4.3 Phase-driven Optimistic Simulation with a Reflection-based Prediction (POSRP)

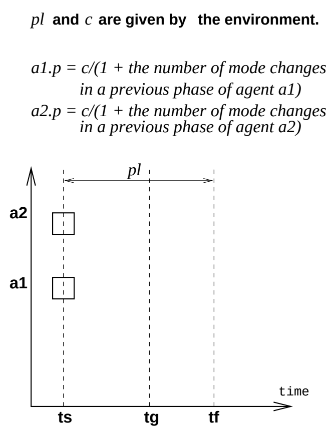
In all three phase-driven algorithms, our goal is to reduce the communication overhead and unnecessary rollbacks without missing any event. In simulating the CHARON agents, the event means mode transitions or invariant violations. The common technique behind all of our phase-driven approaches is that each agent uses approximated dynamics of other agents until they get synchronized. Such an approximation may be a constant or a high-order polynomial. Note that if agents use actual dynamics of shared *read analog* variables which are the input from other agents, instead of using an approximated dynamics, each agent will integrate all such shared *analog* variables controlled by other agents.

Figures 10, 11, 12, and 13 illustrate Phase-driven Optimistic Simulation with a Reflection-based Prediction (POSRP). In POSRP, we simulate the agents with approximated dynamics of shared input variables until an agent reaches the end of the phase *tf* or a warning condition of an agent is satisfied. A warning condition is the disjunction of a guard and a *reflection-based* flag. The rationale behind a warning condition is that if a guard is satisfied, it will cause the abrupt changes of dynamics of the agent. Also, we observe that the approximated dynamics of the variables $appr(v1)$ and $appr(v2)$ are shared information between the agents, where the events are associated with the variables and thus we do not want to use the incorrect value of those input variables.

By examining self-extracted information on the present dynamics or *reflection*, we can flag that agents should synchronize. That is, the agent *a1* stops not only when the guard becomes true but also when $|appr(v1)-v1| >= p$. p is a threshold updated adaptively, based on the history of the mode changes. We update p adaptively, as a small value of p causes unnecessary synchronizations while a large value does more rollbacks. So, we adjust p of the agent a, $a.p$ to be $c/(1$ + the number of mode changes in a previous phase of the agent a). Thus, agents will get synchronized less frequently, if mode changes occur less in a current phase. In case there is no mode change in the current phase, the value of p remains the same in the next phase. c is a parameter given by users.

Figures 10 and 11 depict the setup stage and the computation stage. Figures 12 and 13 show the event notification sub-stage and the rollback sub-stage of the synchronization stage. Note that we call the start time of each phase ts. The final time is called tf. Also, the phase length *pl* is defined as tf - ts.

- Setup Stage (Figure 10): We update p in each agent at time ts. A new p is $c/(1$ + the number of mode changes in a previous phase). Initially, p is the same as c. Then, information to compute the approximated values of shared variables the other agents need are broadcast. We use first-order polynomials as the approximation of dynamics in the current approach. Thus, $a.appr(v)$ is defined as $m * (a.lc - ts) + v_{ts}$, where ts is the start time of the current phase and m is the coefficient of the highest order term of dynamics of v. Each agent transmits m to the other agents. The phase length pl is fixed in the setup stage of $phase_0$. We assume that the initial state is known a priori.
- Computation Stage (Figure 11): Every agent computes dynamics until its local clock reaches tf or the $warning\ condition$ is evaluated to be true. In agent $a1$, the warning condition is $(a1.v1 < 8.0\ and\ |a1.v1 - a1.appr(v2)| > 1.1) \vee (|a1.appr(v1) - a1.v1| >= a1.p)$. $a1.appr(v1)$ is $m1*(a1.lc-ts)+v1_{ts}$ and $a1.appr(v2)$ is $m2*(a1.lc-ts)+v2_{ts}$, where $v1_{ts}$ and $v2_{ts}$ are the values of $v1$ and $v2$ at the start time of the current phase ts and $m1$ and $m2$ are the coefficients of the highest order term of dynamics of $v1$ and $v2$, which are 2.0 in both agents in the example.
- Synchronization Stage (Figures 12 and 13)
  - Event notification sub-stage: Suppose $a2$ reaches tf and the warning condition of $a1$ is satisfied. Then, $a1$ notifies an event to $a2$ with its local clock tg. After $a2$ being informed of an event, it sends the state of $a2$ at time tg to $a1$ that $a1$ can evaluate the guard with the actual value of $v2$.
  - Rollback sub-stage: If $a1$ confirms the event then $a2$ must rollback. After $a2$ rollbacks to time tg, both agents discard the history of the state between ts and tg. At the end of the rollback sub-stage, ts is modified to tg and the next phase starts. In a case that the event was not confirmed by $a1$ with the actual value of $v2$, $a1$ continues its computation.



pl and *c* are given by the environment.
a1.p = c/(1 + the number of mode changes in a previous phase of agent a1)
a2.p = c/(1 + the number of mode changes in a previous phase of agent a2)
pl
a2
a1
time
ts
tg
tf

Figure 10. POSRP Setup stage

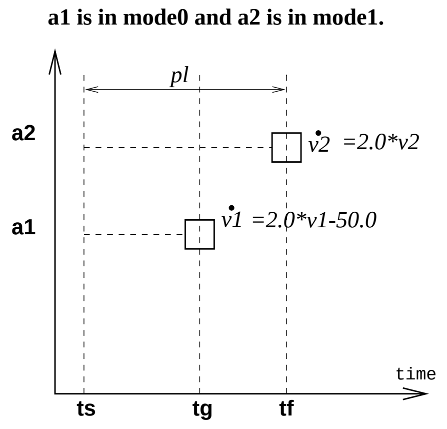


Figure 11. POSRP Computation stage

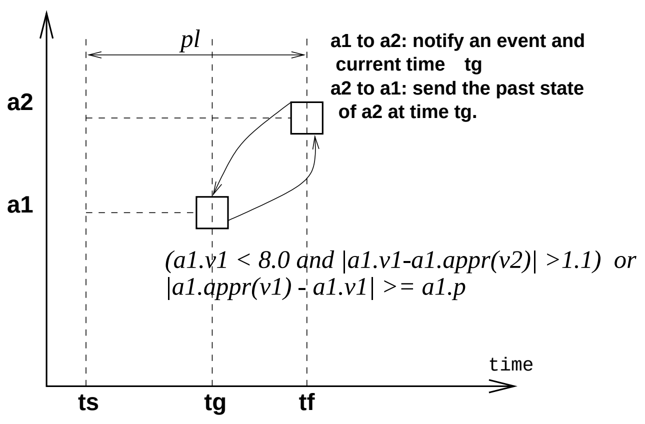

pl
a2
a1
a1 to a2: notify an event and current time tg
a2 to a1: send the past state of a2 at time tg.
(a1.v1 < 8.0 and |a1.v1-a1.appr(v2)| >1.1) or |a1.appr(v1) - a1.v1| >= a1.p
time
ts
tg
tf

Figure 12. POSRP Synchronization stage (event notification)

4.4 Phase-driven Optimistic Simulation with an Adaptive Phase Length (POSAPL)

In POSRP, the phase length pl is given by the simulation environment and fixed in all the simulation phases. However, a large value of pl causes unnecessary rollbacks while a small value of pl will result in frequent synchronizations. In POSAPL, we adjust pl in each phase to enhance the performance. The only difference between POSRP and POSAPL is at a setup stage. If there is no rollback in $phase_{i-1}$ then pl will be doubled in $phase_i$, otherwise it will be reduced to half the length.

4.5 Phase-driven Optimistic Simulation with an Event Predictor (POSEP)

If we know exactly when the next event will occur, we can determine the synchronization point easily. There have

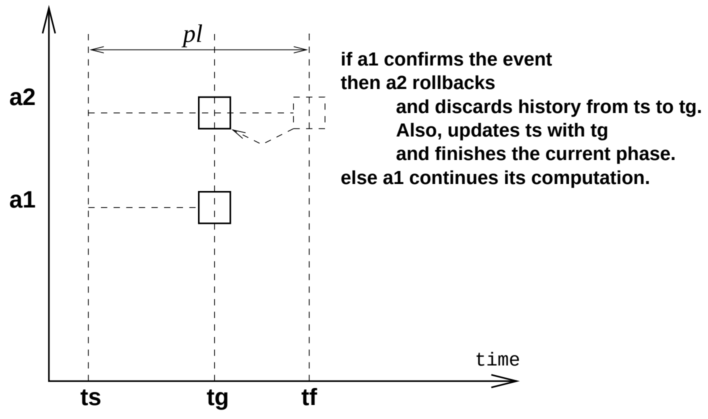

pl
a2
a1
if a1 confirms the event
then a2 rollbacks
and discards history from ts to tg.
Also, updates ts with tg
and finishes the current phase.
else a1 continues its computation.
time
ts
tg
tf

Figure 13. POSRP Synchronization stage (rollback)

been studies on detecting events in differential-algebraic models [16] and hybrid systems [10].

The third method of our optimistic approaches is based on the estimator that predicts when the next event will occur. In POSEP, $Estimator$ is a special process which collects the agent states and the constraints of the active modes from all agents. $Estimator$ solves the constraints to predict tf and broadcasts it with the approximated dynamics of shared variables at the setup stage. Figures 14, 15, 16, and 17 illustrate an optimistic distributed simulation algorithm POSEP.

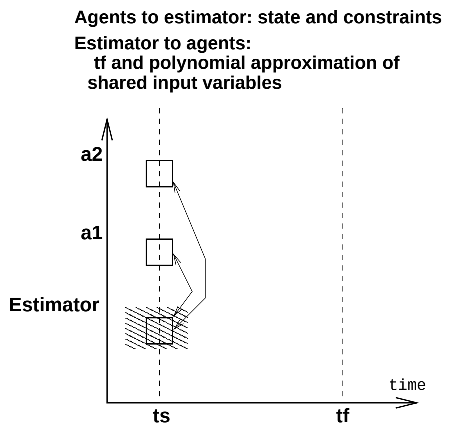

Agents to estimator: state and constraints
Estimator to agents:
tf and polynomial approximation of shared input variables
a2
a1
Estimator
time
ts
tf

Figure 14. POSEP Setup stage

- Setup stage (Figure 14): Agents send their current states and constraints to $Estimator$ at ts. $Estimator$ solves the constraints and predicts when the next event will occur. Such information is sent to the agents with the polynomial approximation of shared variables. Thus, we do not need pl to decide the final time tf of the current phase. $Estimator$ computes and broadcasts tf to the agents $a1$ and $a2$.

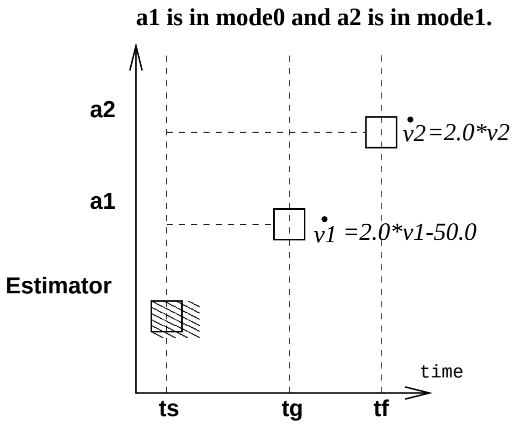
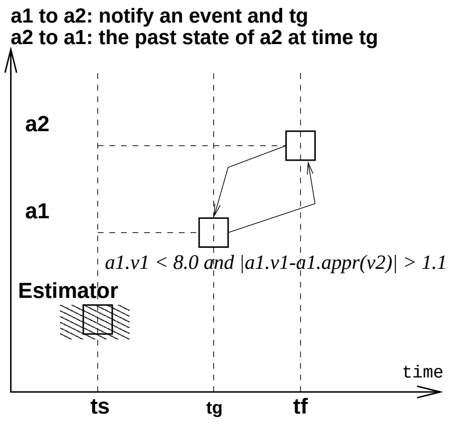


Figure 15. POSEP Computation stage



a1 to a2: notify an event and tg
a2 to a1: the past state of a2 at time tg
a2
a1
a1.v1 < 8.0 and |a1.v1-a1.appr(v2)| > 1.1
Estimator
time
ts
tg
tf

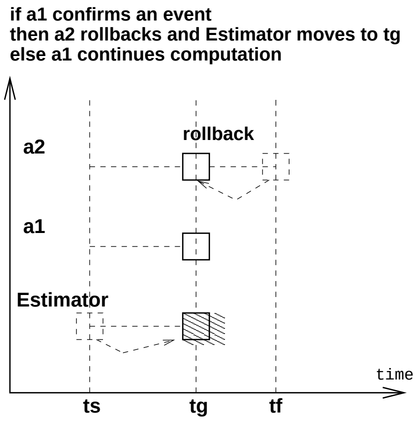
Figure 16. POSEP Synchronization stage (event notification)

- Computation stage (Figure 15): As $Estimator$ provides information on the event occurrences, agents do not have to utilize $reflection$ to predict the possible next event occurrences. So, a warning condition is related to only guards. In the example, the warning condition of $a1$ is $a1.v1 < 8.0$ and $|a1.v1 - a1.appr(v2)| > 1.1$.

- Synchronization stage (Figures 16 and 17):

  – Event notification sub-stage: Suppose $a2$ reaches tf and the warning condition of $a1$ is satisfied. Then, $a1$ notifies an event to $a2$ with the its local clock tg.

  – Rollback sub-stage: If $a1$ confirms the event then $a2$ must rollback. After $a2$ rollbacks to time tg, the both agents discard the history of the state between ts and tg and $Estimator$ process advances up to tg. The current phase ends and ts is updated to tg. As is the case in POSRP, $a1$ will continue its computation if the event is not confirmed by $a1$.



if a1 confirms an event
then a2 rollbacks and Estimator moves to tg
else a1 continues computation
rollback
a2
a1
Estimator
time
ts
tg
tf

Figure 17. POSEP Synchronization stage (rollback)

4.6 Comparison

Table 1 compares different distributed simulations of hybrid systems. A potential speedup from the distributed simulation of hybrid systems depends on the following three factors mainly: communication overhead, rollback overhead, and computation time for numerical integrations and solving constraints.

The communication overhead is extremely large in the algorithm CS (Conservative Simulation), as agents communicate every single update round. In phase-driven optimistic simulations, the communication overhead is relatively small. Contrast to all the optimistic approaches, there is no rollback overhead in the conservative approach. As we use the constraint solver to predict a possible synchronization point better in POSEP, the rollback overhead will be smaller in POSEP than in POSRP or POSAPL. Although there will be the smaller rollback overhead in POSEP, it takes normally a large amount of computation time in solving constraints. Thus, it will depend on the characteristics of given models whether we benefit from POSEP more than POSAPL.

5 Conclusion and Future Work

We have illustrated how to model interactive multi-agent hybrid systems using a modeling language CHARON and proposed four distributed simulation algorithms: one conservative simulation algorithm and three optimistic algorithms. The three optimistic algorithms are phase-driven. In general, the optimistic approaches have the more potential for a speedup than the conservative approach, since agents will not be blocked while waiting for the other agents to finish their phases. Our approaches restrict optimism within each phase to reduce the high rollback overhead caused by a large gap among the local clocks of different agents.

We are implementing the proposed simulation methods over a network of workstations and also on shared-memory multiprocessor machines. Since performance depends largely on how much agents need to communicate, we are currently developing clustering and load balanc-

name	communication overhead	rollback overhead	potential speedup
Conservative	very large	N/A	low
POSRP	relatively small	large	low
POSAPL	relatively small	medium	high
POSEP	relatively small	small	high

Table 1. Comparison of different distributed simulations of hybrid systems

ing techniques for the proposed distributed simulation algorithms.

Acknowledgment. We would like to thank the former and current members of the Hybrid Systems group of the University of Pennsylvania, especially, Rajeev Alur, Rafael Fierro, Radu Grosu, Vijay Kumar, George Pappas, and Oleg Sokolsky for productive discussions in the regular meetings and CHARON developers, Franjo Ivancic, Usa Sammapun and Valentina Sokolskaya for developing the toolset together. This research was supported in part by DARPA ITO MOBIES F33615-00-C-1707, NSF CCR-9988409, NSF CCR-0086147, and NSF CISE-9703220.

References

[1] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In *Embedded Software*, LNCS 2211, T. A. Henzinger and C. M. Kirsch (Eds.), pages 14–31. Springer-Verlag, 2001.

[2] R. Alur, A. K. Das, J. Esposito, R. Fierro, G. Grudic, Y. Hur, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, J. Spletzer, and C. J. Taylor. A framework and architecture for multirobot coordination. In *Experimental Robotics VII*, LNCIS 271 D. Rus and S. Singh (Eds.), pages 303–312. Springer, 2001.

[3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control*, LNCS 1790, N. Lynch and B. Krogh (Eds.), pages 6–19. Springer, 2000.

[4] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 2034, M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli (Eds.), pages 33–48. Springer, 2001.

[5] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[6] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[7] A. Deshpande, A. Gollu, and L. Semenzato. The SHIFT programming language and run-time system for dynamic networks of hybrid automata. Technical Report UCB-ITS-PRR-97-7, University of California at Berkeley, 1997.

[8] H. Elmqvist, F. E. Cellier, and M. Otter. Object-oriented modeling of hybrid systems. In *Proceedings of 1993 European Simulation Symposium*, pages 31–41, 1993.

[9] H. Elmqvist, S. E. Mattsson, and M. Otter. Modelica–The new object-oriented modeling language. In *Proceedings of the 12th European Simulation Multiconference*, pages 127–131, 1998.

[10] J. Esposito, G. Pappas, and V. Kumar. Accurate event detection for simulating hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 2034, M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli (Eds.), pages 204–217. Springer, 2001.

[11] R. Fierro, Y. Hur, I. Lee, and L. Sha. Modeling the Simplex Architecture using CHARON. In *Proceedings of the 21st IEEE Real-Time Systems Symposium WIP Sessions*, pages 77–80, 2000.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Upper Saddle River, NJ, US, 1995.

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

[14] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):99–113, 1989.

[15] S. E. Mattsson and M. Anderson. Omola–An object-oriented modeling language. In *Recent Advances in Computer-Aided Control Systems Engineering*, Studies in Automation and Control 9, M. Jamshidi and C. J. Herget (Eds.), pages 291–310. Elsevier, 1992.

[16] T. Park and P. Barton. State event location in differential-algebraic models. *ACM Transactions on Modelling and Computer Simulation*, 6(2):137–165, 1996.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C, 2nd Ed.* Cambridge University Press, Cambridge, UK, 1992.

[18] J. Steinman. SPEEDED: A unified framework to parallel simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pages 75–83, 1992.