A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation

Peter Frey, Radharamanan Radhakrishnan, Harold W. Carter, *Senior Member*, *IEEE*, Philip A. Wilsey, *Senior Member*, *IEEE*, and Perry Alexander, *Senior Member*, *IEEE*

Abstract—This paper describes a formal framework developed using the Prototype Verification System (PVS) to model and verify distributed simulation kernels based on the Time Warp paradigm. The intent is to provide a common formal base from which domain specific simulators can be modeled, verified, and developed. PVS constructs are developed to represent basic Time Warp constructs. Correctness conditions for Time Warp simulation are identified describing causal ordering of event processing and correct rollback processing. The PVS theorem prover and type-check condition system are then used to verify all correctness conditions. In addition, the paper discusses the framework's reusability and extensibility properties in support of specification and verification of Time Warp extensions and optimizations.

Index Terms—Formal specification, formal verification, theorem proving, parallel discrete event simulation, time warp.

1 Introduction

The *time warp* mechanism is an emerging technique for synchronizing parallel discrete event simulators [12], [15]. In a time warp synchronized simulator, the simulation objects (executing in parallel) exchange time-stamped event messages and execute optimistically—without strict enforcement of the causal order between simulation events. Thus, the protocol permits out-of-order event processing to occur. Whenever such processing does occur, the simulator is forced to rollback and reprocess the events in their correct causal order.

Due to their parallel nature and weak synchronization semantics, time warp simulators are difficult to design and implement. Transient errors or race conditions are difficult to replicate using traditional debugging techniques. Frequently, monitoring code inserted to pinpoint errors causes these errors to disappear. The shear number of simulation events processed by a parallel simulator cause even low probability errors to present themselves. The authors have been working with time warp simulation for over five years and this experience has repeatedly demonstrated these problems and motivated a desire to pursue alternate design approaches.

- *P. Frey is with Cadence Design Systems, 555 River Oaks Parkway, MS 3B1, San Jose, CA 95314. E-mail: pfrey@cadence.com.*
- *R. Radhakrishnan, H.W. Carter and P.A. Wilsey are with the Electrical and Computer &Engineering Computer Science Department, University of Cincinnati, PO Box 210030, Cincinnati, OH 45221-0020. E-mail: {ramanan, hcarter, paw}@ececs.uc.edu.*
- *P. Alexander is with the University of Kansas/Information and Telecommunications Technology Center, 2291 Irving Hill Rd., Lawerence, KS 66044. E-mail: alex@ittc.ukans.edu.*

Manuscript received 01 Feb. 1999; revised 13 Aug. 2000; accepted 02 May 2001.
Recommended for acceptance by A.M.K. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109094.

A potential solution to these problems is provided by formal specification and verification. The application of this software engineering methodology helps eliminate ambiguities in the system description during the specification process and supports mechanized verification of critical properties. Although the manpower costs of applying formal methods can be quite high, the anticipated resource savings during debugging, the wide use of the modeled simulation kernel, and the reusability of the framework justify this expense.

The objective of this research effort is the development of a formal, extensible framework that facilitates verification and exploration of the time warp simulation protocol. This encompasses three tasks: 1) formal specification of the basic time warp protocol, 2) formal verification of the time warp specification, and 3) generation of a reusable specification framework for use in time warp simulator research and development. The formal specification goal is achieved by developing and specification of a basic time warp simulation algorithm in the PVS specification language. The verification goal is achieved by identifying correctness conditions and using PVS to prove those correctness conditions. Finally, the reusability goal is achieved by judicious use of the PVS type-checking condition system to automatically generate verification conditions and by avoiding details of any specific simulation system in the specification.

Similar endeavors with mathematical proofs of parallel discrete event simulation (PDES) algorithms have been carried out by other researchers. Mathematical studies have been conducted by Ghosh [13] who presents a proof of correctness for the YADDES asynchronous distributed algorithm. Lin [19] presents an algorithm for determining the global progress of a parallel simulation with a FIFO communication property and the proof of correctness for

this algorithm. In his PhD thesis, Bauer [3] presents a proof of correctness for a novel distributed snapshot determination algorithm which he proposes. Kannikeswaran et al. have also formally specified and verified another distributed snapshot determination algorithm [16]. These efforts primarily focused on specific aspects of a parallel and distributed simulation system. Leivent and Watro [18] were the first to suggest a mathematical foundation for a time warp-based system. They developed a simple formal model of the the time warp approach to distributed computation and proven several important properties of the model. In addition, they used their model to devise some extensions to the time warp algorithm that provides improved termination behavior. Gopalakrishnan and Fujimoto [14] use formal methods to prove correctness properties of a hardware chip to support rollback processing for Time Warp simulation. Their work does not develop a complete specification of Time Warp, but focuses on verifying the functional behaviors of a particular hardware design.

The formal specification framework presented here is similar to the mathematical framework suggested by Leivent and Watro [18] in that it also models a complete time warp system. The distinction between Leivent and Watro and this work is the use of a software specification language rather than a purely mathematical model. Since the PVS specification is closer to the actual implementation than a mathematical model, it assists the software engineer in the process of mapping desired properties onto implemented subsystems and reduces the semantic gap between the specification and implementation. Further, the PVS verification system enables the software engineer to mechanically verify specific properties and, in many cases, automatically reverify properties following design modifications.

The paper continues with a brief introduction to PVS [23], [25] in Section 2. (Readers familiar with PVS may skip this section.) Section 3 then details the basic time warp protocol as modeled here. The mapping of the individual building blocks of the time warp protocol onto their formal representation is presented in Section 4. The verification process is then described in Section 5. The conclusions, presented in Section 6, reevaluate the formal representation of the time warp paradigm and the applicability of the formal framework to other research and development avenues.

2 Formal Specification and Verification in PVS

PVS provides a specification language, a powerful theorem prover, supportive tools, and basic definitions in libraries and the prelude file. PVS is specifically constructed to detect errors in the early stages of system design. Specifically, PVS has three features that allow it to detect such errors [7], [31]:

- The PVS specification language is based on a typed higher-order logic. PVS provides a rich set of types and the ability to define subtypes and dependent types.
- PVS supports specification using a form of *conservative extension* that ensures consistency of specifications. Specifically, if a consistent theory is extended in a conservative manner, the resulting theory is also consistent.
- The PVS theorem prover provides a powerful, extensible system for verifying obligations. The strength of the theorem prover comes from the set of inference and decision procedures native to the proof environment. Proofs result in scripts that can be edited, attached to additional formulas, and rerun.

To aid in understanding subsequent specifications, the structure of an example PVS specification is illustrated in Table 1. A specification is formed by combining theories describing various components and properties. Each theory is partitioned into assumptions, definitions, axioms, and theorems. Table 1 shows an example of the two basic specification styles, property-based and conservative extension. Property-based specification represents the most general style. A declaration of the signature is presented in the definitions section followed by separate axioms defining function and constraints. In Table 1, `push`, `pop`, and `top` are operations defined using the property-based style. Although property-based specification is maximally flexible, it is difficult in most situations to assure that resulting specification is consistent. In contrast, `empty?` and `nonempty?` are defined using conservative extension to avoid introducing inconsistencies.

2.1 Conservative Extension

Extension, the process of adding definitions to an existing theory, is a common mechanism for writing specifications. A set of base definitions including types and primitive operations is extended to define new types and operations. In a monotonic logic like PVS, every consequence of the original theory is a consequence of the new theory. Because none of the original definitions are removed, any consequence of the original theory is still derivable. New consequences result from the added definitions, thus extending the theory to define a new system.

One theory is a conservative extension of another if the extension adds axioms that only define properties over new operations. Specifically, new axioms define properties of new operations and do not define new properties of existing operations. By adding properties for new operations only, potentially inconsistent properties of existing operations cannot be added. Thus, if a consistent theory is extended in a conservative manner, the resulting theory is guaranteed to be consistent. Because it is difficult to check ad hoc extensions for consistency, conservative extension is used exclusively in the time warp specification.

PVS provides a definition style for operations that guarantees a conservative extension [26]. This PVS shorthand defines a signature for the new function and adds a single axiom which equates the new signature with a PVS expression. As an example, the specification of an increment function, `empty?(s:S):bool = (s=empty)`, follows the conservative extension style. The signature part defines `empty?` as a predicate from stack to Boolean, while the definition asserts that `empty?(x)` is equal to `s=empty`. If all theory extensions introduce a new function and a single defining axiom to an already consistent theory, then

TABLE 1
Example Specification

```
Stack[S,T:TYPE+]: THEORY
BEGIN
%% Assumptions
  ASSUMING
  %% Properties assumed on parameters
  ENDASSUMING
%% Declarations
  empty: S
  nonempty?(s:S):bool = (s/=empty)
  empty?(s:S):bool = (s=empty)
  push: [T,S -> (nonempty?)]
  pop: [(nonempty?) -> S]
  top: [(nonempty?) -> T]
%% Axioms
  push_nonempty: AXIOM (forall (s:(nonempty?)): push(top(s),pop(s))=s)
  push_empty: AXIOM (forall (s:S,t:T): push(t,s)/=empty)
  ...
%% Theorems
  empty?_empty: THEOREM empty?(empty)
  ...
END Stack
```

the resulting specifications are consistent. This eliminates the difficult task of determining consistency of theories.

Additionally, PVS uses, conservative extension function definition to define rewrite rules that enable powerful automated verification. Thus, using conservative extension not only guarantees consistency, but simplifies the proof process by supporting automated rewriting.

2.2 Predicate Subtypes and TTCs

Of particular importance to this work is the use of PVS predicate subtypes and automatic generation of type-check conditions (TCCs). Predicate subtypes provide a mechanism for defining new types using comprehension. Given any predicate, `s?`, with domain, `D`, the predicate subtype `(s?)` is defined as $\{d : D \mid s?(d)\}$. The function signature `f(d:(s?)):(s?)` defines a function whose input and output must satisfy the `s?` predicate. Using predicate subtypes in this way defines `s?` as an invariant property over `f`.

Using predicate subtypes, it is possible to use the PVS type checking system to automatically generate and verify obligations. Two type check conditions must be satisfied by any application of `f(d)`: 1) the operation must produce a value satisfying `s?` and 2) each instantiation of the parameter `d` must satisfy `s?`. The former obligation is checked once for the definition while the latter must be checked each time the operation is used. When used in this manner, `s?` becomes an invariant over `f`. Using this specification approach is advantageous because PVS will attempt to prove the invariant property automatically as a part of the normal type checking process. If the type checking system cannot determine whether an operation is type correct, it automatically generates a type check condition (TCC) that must be proven interactively by the user.

As an illustrative example, consider the expression `if x then f(y) else f(z) endif;`. One type check condition is generated for each application of `f`: 1) `x implies s?(y)` and 2) `forall not(x) implies s?(z)`. Each TCC checks that the actual parameter of `f` is an element of the predicate subtype defining the domain of `f`. Note that each TCC places the test in the context of the respective application of `f`. Specifically, `f(y)` occurs when `x` is true, while `f(z)` occurs when `x` is false. For more details on the PVS type checking system, please refer to the PVS documentation [24].

2.3 Notation and Terminology

References to axioms, type definitions, and theorems in the description are printed using the `Typewriter` font style. Identifiers printed with the Sans Serif font style represent theories comprising the specification. All operators and axioms defined in this specification are introduced using conservative extension, thus readers will not see the traditional `AXIOM` identifier in this specification. Conservative extension definitions can be identified by their general structure equating a signature with an expression. The term `CONJECTURE` is used exclusively in this specification to identify theorems. Note that all such theorems are used to discharge TCCs generated during verification. Predicates defining properties of objects are given names ending in "`?`" as is traditional in PVS.

Other PVS specific notations used extensively in the time warp specification include `LAMBDA`, `every`, and `WITH`. The `LAMBDA` notation allows the definition of unnamed functions as is traditional in many languages. The `every` operation provides a shorthand notation for universal quantification over predicate subtypes. Specifically,

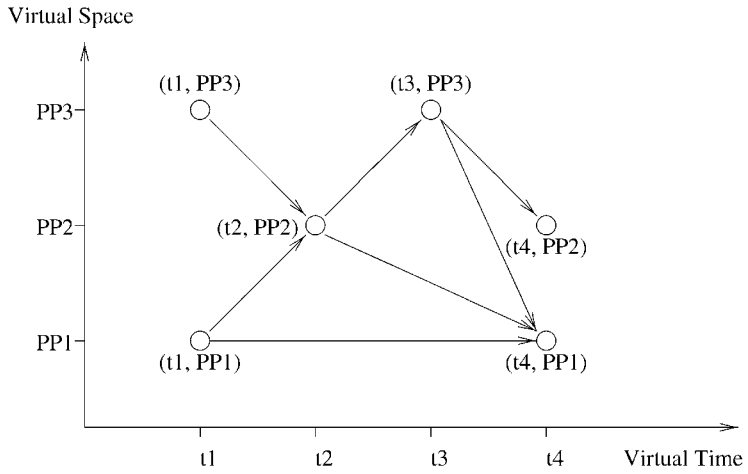

Fig. 1. Simulation trace in a virtual coordinate system.

`every(p)(a):bool = FORALL (x:(a)):p(x)`, where `a` is a predicate used to define a subtype of its domain and `p` is a predicate used to test each element of the subtype. Finally, `WITH` is an override operation used to modify records and tuples. The notation `R WITH f:=v` produces a copy of the record `R` with value `v` in field `f`.

3 The Time Warp Protocol

This section serves as an introduction to the problem domain of parallel discrete-event simulation (PDES), specifically the parallel optimistic time warp paradigm. Common terminology, as used by the PDES community, and principles of the algorithm are introduced to assist in the comprehension of the formal specification and verification approach. The clarity and structure of the description is an actual result of the formal specification presented in Section 4.

Discrete-event simulation (DES) is a method of simulation where changes to a system are modeled as an ordered sequence of actions. In DES, the model to be simulated is decomposed into a set of physical processes.[1] Physical processes interact with each other by exchanging messages and the arrival of messages determines the order of actions to be performed. Parallel discrete event simulation (PDES) is concerned with maintaining the same ordered execution in a distributed environment, i.e., the sequence of correct execution steps taken by a parallel simulation must be the same as that of a sequential simulation. One mechanism for achieving the required ordered execution (or synchronization) on a distributed platform is time warp [15].

Execution order is defined using the notion of a simulation clock that advances in simulation time. Simulation time is a one dimensional, temporal coordinate system used to measure computational progress and to define synchronization. Simulation time can have infinitely many positive values. Depending on the model to be simulated, simulation time may be defined using several attributes; simulation time can be discrete or continuous, may or may not be totally ordered, and may or may not be related to real time. Any one or a combination of these time attributes can be abstractly defined as *virtual time*. We define virtual time as a unit of measure as opposed to Jefferson's definition, where he introduces Virtual Time as a paradigm [15].

In PDES, the model to be simulated is usually composed of several physical processes (PP) that are executing concurrently on a uni/multiprocessor system. Processes are uniquely identified by their *process identifier*. The set of all process identifiers constitute a one dimensional spatial coordinate called the *virtual space* of the system. Given a virtual time (t) and a virtual space coordinate (pp), a *virtual coordinate* system (t, pp) can now be defined. Any specific action or a set of actions can be identified to have occurred at a specific virtual coordinate during the course of the simulation. The sequence of actions that constitute a simulation can be mapped to specific points in the virtual coordinate system. If the model to be simulated is deterministic in nature, then repeated simulations will always map to the same set of points in the virtual coordinate system. Fig. 1 illustrates a trace of the simulation in the virtual coordinate system. Actions are depicted as vertices in the graph. The directed edges in the graph represent the temporal relationship between the actions.

Temporal relationships (edges) between actions are defined by *messages* in a simulation model. Two virtual coordinates are associated with every message. The first virtual coordinate is used to identify the sender and the virtual send time of the message. The second virtual coordinate identifies the receiver and the virtual receive time of the message. Two fundamental rules, known as Lamport's Clock Conditions [17], are enforced on any two adjacent virtual coordinates:

- **Rule 1.** The virtual send time of each message must be less than its virtual receive time.
- **Rule 2.** The virtual time of each event at a process must be less than the virtual time of the next event at that process.

A single vertex in the graph represents a set of actions on a specific physical process at a specific instance in virtual time. This set of actions determines the behavior of a physical process. In general, the set of actions that determines the process's behavior is represented as an

1. By "physical" processes, we mean the processes that model the components of the real world entity (where physical implies the real world).

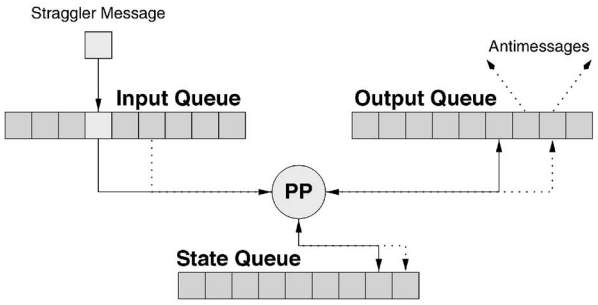

Fig. 2. Structure of a simulation object.

ordinary deterministic computation. It involves zero or more of the following operations:

1. receive any number of messages stamped with receiver *pp* and virtual receive time t, and read their contents,
2. read its current virtual time instance,
3. update its state variables, and
4. send any number of messages, all of which will automatically be stamped with sender *pp* and virtual send time t.

In addition, simulation paradigms often include additional information along with the message. The combination of this additional information and the message is often referred to as an *event*. This definition of event differs from Jefferson's original definition [15]. In the case of time warp, an event contains a sign field in addition to the virtual time coordinates and the message body (message content).

The physical process description is mapped into the simulation paradigm with the help of additional support structures. The physical process and the additional structures are collectively known as a *simulation object* or as a *logical process*. Fig. 2 illustrates the simulation object and identifies the additional structures (three queues) required by the time warp paradigm. Incoming events are stored in the *input queue* and outgoing events are stored in the *output queue*. The state of the physical process, along with the current virtual time instance of the process, are stored in the *state queue*. This instance of virtual time associated with the process's state is called the *local virtual time* (LVT) of the simulation object.

The time warp protocol is required to correctly handle event arrival, event processing, state saving, and event departure, thereby enforcing the correct sequence of actions. As each simulation object in the set simulates asynchronously, it is possible for the object to receive an event from the past (some simulation objects will be processing faster than others and, hence, will have LVTs greater than others) violating the causality constraints of the events in the simulation. Such events are called *straggler* events. On receipt of a straggler event, the simulation object must rollback to an earlier state, undoing some work that has been done. Rollback involves two steps: 1) restoring the state to a time preceding the timestamp of the straggler and 2) canceling any output events that were erroneously sent (by sending *antimessages*, copies of the original erroneously sent event but with their sign inverted). Erroneous events are removed from the input queues of recipient simulation objects by comparing them with the incoming antimessages. This process is called *event annihilation*. After rollback, the events are reexecuted in the proper causal order.

For rollback recovery to take place correctly, state needs to be restored. In the original time warp algorithm, state is saved after every event execution. While this entails a significant overhead in terms of memory consumed, it is necessary to protect against erroneous optimistic computation. Each simulation object must periodically save its local state such that, in the event of a causality error, a rollback to a correct state is possible. Time warp objects with large states require considerable memory space as well as CPU cycles for state saving. In general, states are saved after every event execution. However, the check-pointing cost can be reduced by saving the state infrequently. In the simple case, a time warp simulator checkpoints every χ events; this scheme is called *periodic* or *infrequent* state saving [8], [29].

Having described the steps involved in the execution of an asynchronous set of distributed processes, it is established that there is no global control among these processes. However, to determine simulation progress or detect termination of the simulation, a global consensus must be achieved. For this reason, Global Virtual time (GVT) is defined. GVT is a property of an instantaneous global snapshot of the system at real time r and is defined as follows:

Definition 1. *GVT at real time r is the minimum of 1) all virtual times in all virtual clocks at time r and 2) of the virtual send times of all messages that have been sent but have not yet been processed at time r.*

Given this definition of GVT, it can be inferred that GVT must *never decrease*. GVT serves as a floor for the virtual times that any process can ever rollback to. In fact, if every event completes normally, if messages are delivered reliably, if the scheduler does not indefinitely postpone the execution of the farthest-behind processes, and if there is sufficient memory, then GVT *must eventually increase* [16]. Instantaneous values of GVT are impossible to compute in a distributed system. Hence, several methods to accurately estimate GVT have been proposed in the literature [4], [5], [20], [22]. Estimates of GVT are used for termination detection, memory management by reclaiming (or fossil collecting) old states and events, error handling, and committing input/output operations.

4 Formal Specification of the Time-Warp Protocol

Having introduced the time warp paradigm, this section details the mapping of the basic time warp algorithm to a formal specification in the Prototype Verification System (PVS) [23], [25]. The specification of time warp is hierarchically organized. Fig. 3 illustrates the organization of the theories provided in the specification. An edge points to the theory that includes the originating theory. Fig. 3 presents an additional theory called Sequential (in a dashed box). This theory is independent from the top-level specification of the time warp simulator and represents the specification

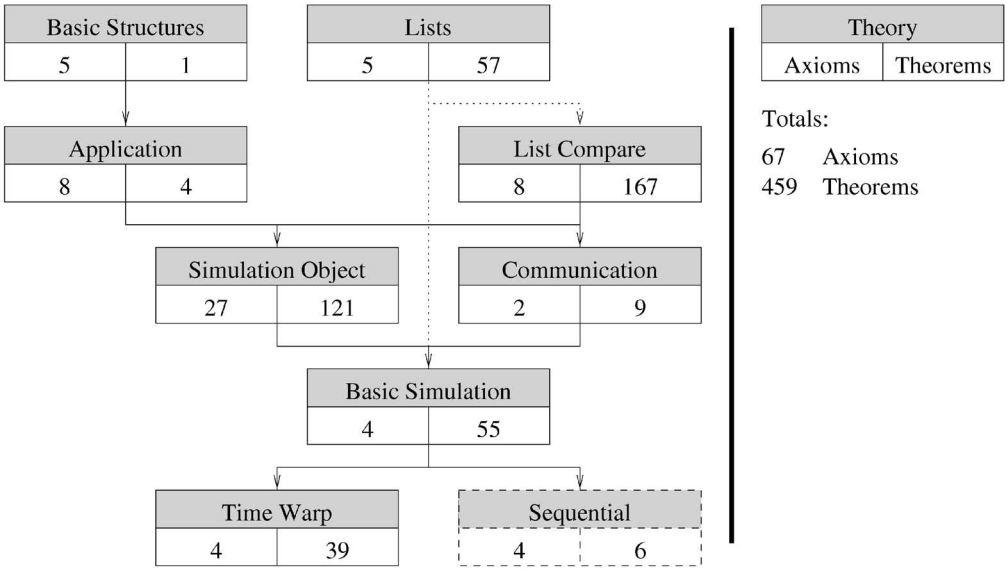

Fig. 3. Hierarchy of theories.

of a sequential simulation kernel. It is only included here to show how the provided axioms are reused to specify a sequential simulation paradigm.

Conformance to software engineering aspects, such as reusability and extensionality, are the main issues responsible for the current specification structure. Lists and List Compare provide the major time warp list functionalities and are parameterized theories. Theory Lists is parameterized exclusively on the list element type while List Compare requires a mapping function from a list element to a real value used in the ordering relation. The parameterized theories enable the specification to be used as state and event lists in the time warp structure (Fig. 2) and as message buffers in the simple communication layer. Re-usability of the specification is also the reason for the existence of the Basic Simulation theory. This theory provides axioms common to the specification of time warp and the sequential simulation protocol.

We abstract the discrete-event simulation paradigm from the application by using parameterized theories and by using the *importing* statement of PVS. Time warp structures that require information from the application are specified in the Basic Structures theory. This parameterized theory is used by the Application theory which provides `Time`, `StateVariables`, `MsgBody`, and `ProcessId`. Importing the Application theory will therefore provide the complete specification of events and state to the time warp/sequential simulation environment. Thus, the use of hierarchical specification results in an exact interface specification between the simulation model and the simulation kernel. In fact, a similar approach is used in the design of an object-oriented time warp simulation kernel [28].

In addition, Fig. 3 presents the number of axioms and theorems defined in each theory as well as their totals. Every declaration (type, operation and axiom) is counted as an axiom. This includes operations defined using conservative extension. The number of axioms are illustrated in the figure to give the reader an idea of the size of the specification and the expressive power of the axioms. A major goal during specification is to keep the number of axioms low and as flexible as possible. This enhances their reusability and helps to keep the number of supporting theorems low.

4.1 Axioms and Type Definitions

The first theory to be introduced is the Application theory that contains application related type information. As no specific discrete event model is intended as an application, generic types are introduced to represent various time warp quantities (see Table 2).

The Application theory maps the type `Time` (virtual time coordinate) onto real numbers. Using real numbers as the virtual time scale ensures that the specification is valid for a

TABLE 2
Application Theory—Some Basic Types

```
Time : TYPE+ = real
ProcessId : TYPE+ = nat
StateVariables : TYPE+
stateVariables : StateVariables
MsgBody : TYPE+
msgBody : MsgBody
```

TABLE 3
Basic Structures Theory

```
State        : TYPE+
  = [# lVT         : Time,
       stateVars   : StateVariables #]
anState: State

Sign : TYPE = { POSITIVE, NEGATIVE }

Event : TYPE+
  = [# recvTime    : Time,
       sendTime    : Time,
       destination : ProcessId,
       source      : ProcessId,
       sign        : Sign,
       body        : MsgBody #]
anEvent: Event
```

wide variety of actual real-world models that typically require different virtual times [1], [2], [10], [30]. A simpler mapping was chosen for the virtual space coordinate, `ProcessId`. As process identification requires only a large enough set of discrete and unique values, `ProcessId` is mapped onto the set of natural numbers. New types are introduced for physical process state (`StateVariables`) and message content (`MsgBody`). Both of these types are nonempty as it can be safely stated that any discrete event model has at least one state and some messages.

Given the definition of virtual coordinates, process state, and message content, the time warp types `State` and `Event` are defined in the Basic Structures theory (see Table 3). The physical process state `StateVariables`, along with the process' local virtual time constitute the simulation object's state (`State`). The definition of an event requires making the distinction between actual messages and antimessages. The enumeration type `Sign` introduces exactly two possible "signs" for this purpose. A positive sign represents an event that has been optimistically sent while a negative sign represents an antimessage sent to destroy its positive counterpart during rollback. In addition to sign (`sign`) and message content (`body`), an event contains the virtual time coordinates of the sender (`source, sendTime`) and the receiver (`destination, recvTime`).

As stated earlier, adherence to Lamport's clock conditions is required for a correct discrete event simulation. Lamport's first rule is represented in Table 4 as `ValidEvent?`. This predicate is true when the `recvTime` of an event is greater than its `sendTime`. The physical process behavior is defined over a nonempty list of valid events and a state variable. A nonempty list is required because *event driven* execution mandates that at least one event is present for the process to be scheduled for execution. This event also enables the process to access its new current virtual time (LVT) and process id. There can be any number of events for a process at a specific virtual time. Process behavior is modeled by two separate types: 1) `Process` represents state changes and 2) `ProcessOutput` specifies output events generated. Any physical process model can be represented by these two types. Lamport's second clock condition and the virtual coordinates associated with the generated output events is incorporated in the predicate `ValidProcess?`. Generated output events are guaranteed to possess send coordinates equal to the process's current virtual coordinates (process id, LVT). In addition, as processes can send events to themselves (intraprocess communication), Lamport's second clock condition mandates that the `recvTime` of generated output events be greater than the process's current LVT. The process behavior and subtype definitions are also defined in the Application theory (Table 4).

Having defined the three major components of the time warp paradigm (process, process state, and event), the Simulation Object theory (Table 5) provides the specification of a basic simulation object (depicted in Fig. 2). A snapshot of a simulation object is represented by the type `SimObjState`. At any given virtual time, a simulation object can be identified by the state of the input queue (`inputList`), the output queue (`outputList`), the state queue (`stateList`), the process behavior (`process, processOutput`), the current state (`currentProcessState`), and the process id (`id`). Since the simulation object is envisioned to operate in a distributed environment, a local communication buffer (`ether`) was introduced into the simulation object's specification.

Two basic properties are known about the state history in the `SimObjState` and the `current-Process-State`: 1) at any virtual time, a simulation object has stored at least one state in its `stateList` and 2) as physical processes advance in virtual time, newly generated states (i.e., the `currentProcessState`) must contain a virtual time greater than or equal to the LVT of the latest state in the history. The first property is an axiom as at least one "initial" state is present in the `stateList` at the start of the simulation. The second property must hold as state saving occurs only after the execution of a process. The two properties are incorporated into the type predicate `AtLeastAState?` that has to be satisfied by the time warp algorithm.

TABLE 4
Application Theory

```
ValidEvent?(event:Event): bool = recvTime(event) > sendTime(event)

Process : TYPE+ = [ (cons?[(ValidEvent?)]), StateVariables
                    -> StateVariables ]

ProcessOutput : TYPE+ = [ (cons?[(ValidEvent?)]), StateVariables
                          -> list[(ValidEvent?)] ]

ValidProcess?(process:ProcessOutput) : boolean =
(FORALL (state:StateVariables,
          inputs:(cons?[(ValidEvent?)]),
          outputEvent:  (ValidEvent?) ):
      member(outputEvent, process(inputs, state))
      => ((FORALL (inputEvent:(ValidEvent?)):
                  member(inputEvent, inputs)
            => recvTime(outputEvent) > recvTime(inputEvent)
        )
        AND sendTime(outputEvent) = recvTime(car(inputs))
        AND source(outputEvent) = destination(car(inputs))
      )
)
```

TABLE 5
Simulation Object Theory

```
InputList  : TYPE = list[(ValidEvent?)]
OutputList : TYPE = list[(ValidEvent?)]
StateList  : TYPE = list[State]

SimObjState : TYPE
  = [#  inputList    : InputList,
        outputList   : OutputList,
        stateList    : StateList,
        process      : Process,
        currentProcessState : State,
        processOutput: (ValidProcess?),
        id           : ProcessId,
        ether        : list[(ValidEvent?)]
    #]

AtLeastAState?(state:SimObjState) : bool =
  cons?[State](stateList(state))
  AND
  lVT(currentProcessState(state)) >=  lVT(maximum(stateList(state)))
```

The functionality of the time warp queues is specified by the List Compare and Lists theories. Section 3 discusses the importance of the time warp queue structures. However, the term queue is a misnomer as these queues do not exhibit traditional queue properties (such as FIFO or LIFO). As a result, this specification uses the more generic term, "list" instead of queue. The time warp list specification requires a set of definitions extending the list definition provided by PVS. These definitions are provided in the List Compare and the Lists theories. Table 6 only illustrates the interface definitions of these additional definitions as their complete definition is lengthy and is not required for this discussion.

Before the individual definitions can be explained, it is important to note that no restrictions are placed on the elements of a list. In particular, elements can occur more than once in a list. This is an important property of a time warp list. Definitions such as `number`, `sublist`, and `permutation` are required because of this repeated

TABLE 6
List Compare Theory and Lists Theory

```
number(element: T, list:list[T]) : RECURSIVE nat
sublist(list1 : list[T], list2: list[T]) : bool
permutation(list1 : list[T], list2 : list[T]) : bool
removeEqual(element: T, list:list[T]) : RECURSIVE list[T]
remove(list: list[T], list2: list[T]) : RECURSIVE list[T]

getNext(element:T, list:list[T]) : RECURSIVE T
getPrevious(element:T, list:list[T]) : RECURSIVE T
smallest(list:(cons?[T])) : RECURSIVE T
maximum(list:(cons?[T])) : RECURSIVE T
expungeAfter(element:T, list:list[T]) : RECURSIVE list[T]
expungeBefore(element:T, list:list[T]) : RECURSIVE list[T]
getAllAt(element:T, list:list[T]) : RECURSIVE list[T]
putAll(list1:list[T], list2:list[T]) : list[T]
```

TABLE 7
Simulation Object Theory

```
signInsert(event1: (ValidEvent?), list:list[(ValidEvent?)])
signInsertList(newEvents: list[(ValidEvent?)], list: list[(ValidEvent?)])

signInsertProp : CONJECTURE
(
  FORALL (event: (ValidEvent?),
          list: list[(ValidEvent?)]):
  signInsert(event, list) = putAll(list,cons(event,null))
  OR
  signInsert(event, list) = removeEqual( event WITH [ sign := NEGATIVE ], list)
  OR
  signInsert(event, list) = removeEqual( event WITH [ sign := POSITIVE ], list)
)
```

occurrence property. The predicate `member` (defined in the PVS prelude) is true or false depending on the availability of an element in a list. The `number` function has a similar interface to `member`. But, it's range is the set of natural numbers which represents the number of occurrences of a given element. With the help of `number`, it is now possible to define a `sublist` and the `permutation` relation. The `sublist` predicate is satisfied if every member of the first list is present in the second list at least once. If the `sublist` operation is reflexive over the given list, the predicate `permutation` is true and the lists are permutations of each other. Two operations define the removal of elements. `removeEqual` specifies the action of removing a single element from a list, whereas `remove` defines the action of removing a list from another list. In both cases, if an element is not found in the list, the original list is left unchanged. The last of the general list properties used in the time warp specification is the merger of two lists. This functionality is specified by the `putAll` operation definition. This definition renames the `append` operation provided by the PVS prelude.

The seven remaining list operations require the ordering relation defined on list elements. `smallest` and `maximum` are simple observers for the smallest and the largest element of a nonempty list respectively. A similar functionality is provided by `getNext` and `getPrevious`. Given an element, these operations define order relationships between the given element and a specific element in the list using their respective time stamps. If an element larger than the given element is in the list, the `getNext` operation guarantees that the next larger element in the list will be returned. If no larger element exists, the given element is returned. The operations `expungeBefore` and `expungeAfter` are provided to remove elements prior to a particular time (for fossil collection) or everything later (for rollback). All elements indistinguishable by the ordering relation (e.g., events with the same receive time) in a list can be accessed using the `getAllAt` operator.

The input list of a physical process has an additional property, namely, that of *sign dependent insertion*. As this property is only true for the input list of events, the specification is part of the Simulation Object theory (see Table 7). `signInsert` is defined on a single element and then for a list of events to be inserted (`signInsertList`). `signInsert` does not always increase the number of elements in the list. When a positive message meets a

TABLE 8
Simulation Object Theory

```
execute(currentState:(StateAndNewInput?)):(AtLeastAState?) =
  currentState with [
    outputList := putAll( outputList(currentState),
                          processOutput(currentState)(
                            currentInputEvents(currentState),
                            stateVars(currentProcessState(currentState))
                          )
                        ),
    stateList  :=
      cons( anState with [
              lVT := newlVT(currentState),
              stateVars := process(currentState)(
                             currentInputEvents(currentState),
                             stateVars(currentProcessState(currentState))
                           )
            ],
            stateList(currentState)
          ),
    currentProcessState :=
      anState with [
              lVT := newlVT(currentState),
              stateVars := process(currentState)(
                             currentInputEvents(currentState),
                             stateVars(currentProcessState(currentState))
                           )
            ],
    ether      := putAll( ether(currentState),
                          processOutput(currentState)(
                            currentInputEvents(currentState),
                            stateVars(currentProcessState(currentState))
                          )
                        )
  ]
```

negative message, they cancel each other (i.e., annihilate each other) and, hence, it is possible that the number of elements in the list may actually decrease. This property is stated by the `signInsertProp` conjecture. The conjecture does not state that individual cases are mutually exclusive nor that these are the only possibilities.

The axioms and type definitions described thus far specify all the basic building blocks and the structure of a simulation object. The following discussion deals with the functionality of a simulation object. `execute` and `insertEventsCheck` are two operations provided by the Simulation Object theory (see Table 8) that define the actual behavior of a simulation object.

Event driven execution, represented by the `execute` operation, requires pending events. This requirement, is represented in the parameter definition of `currentState` using the type predicate `StateAndNewInput?`. In addition, `StateAndNewInput?` requires that the `AtLeastAState?` predicate be satisfied. The signature of `execute` requires that, at the end of the simulation execution cycle, the `AtLeastAState?` predicate stays satisfied. The `execute` operation specifies how the simulation object's state changes. Only the fields that change during the execution cycle are stated inside the definition's "with" clause. This implies that the execution of a simulation object modifies only the fields of `outputList`, `stateList`, `currentProcessState`, and `ether`. Events generated during the execution of the simulation model are inserted into the fields of `outputList` and `ether`. This models the storage of the events (for rollback processing) in the `outputList` and the process of sending the newly generated messages (by inserting events into the local communication buffer `ether`). The `currentProcessState` field contains the newly generated state consisting of updated process variables and the new virtual time of the process. As a frequent state saving strategy is specified, every execution cycle terminates with the storage of the current state in the `stateList`.

Two rewrite rules are used to specify the `execute` operation. Access to events scheduled for execution is

TABLE 9
Simulation Object Theory

```
currentInputEvents(currentState:(StateAndNewInput?))
  : (cons?[(ValidEvent?)]) =
  getAllAt( getNext( dummyRecv(lVT(currentProcessState(currentState))),
                     inputList(currentState)),
            inputList(currentState)
          )

newlVT(currentState:(StateAndNewInput?)) : Time =
  recvTime(getNext( dummyRecv(lVT(currentProcessState(currentState))),
                    inputList(currentState)
                  )
          )
```

specified by the currentInputEvents rule and newlVT specifies how to determine the new local virtual time of the process. Table 9 illustrates both of these rules. DummyRecv (used in the both currentInputEvents and newlVT rules) specifies an arbitrary event that has a virtual receive time provided in the argument. The generation of an arbitrary event is required as the general time warp list operations work on elements only.

In the case of time warp simulation, defining the procedure for inserting incoming events into the simulation object's input list is critical. The specification of event insertion is provided by the definition of insert EventsCheck and is illustrated in Table 10. If there are no events to be inserted, then a null operation is said to take place and the simulation object's state does not change. This behavior is stated for the sake of completeness. Depending on the time of the smallest incoming event, there are two cases of event insertion: 1) in the future or 2) at the current time or in the past of the current process. In the case that the smallest event is received in the future, it follows that all events are received in the future. As a result, all elements are inserted into the input list of the simulation object. However, event insertion does not necessarily lead to a longer list. As this insertion is sign dependent, annihilation of opposite signs may occur leading to a smaller list of events. In the second case, the smallest event has a time stamp less than or equal to the current time of the simulation object. This causes a causality error and a rollback is initiated. Rollback changes input, state, and output lists as well as the ether of the simulation object state. Even though there are two cases to be handled, the specification identifies the behavior of the insertion to be the same. In both cases, signInsertList is called.

In the case of the state list, two separate actions must occur to establish the simulation object's state before the erroneous computation started. First, the last correct state has to be determined. This is specified in stateBefore. Then, all incorrectly generated states have to be removed. This is specified by the expungeAfter operation. outputlist and ether are modified in a similar fashion. Copies of the erroneously sent messages are retrieved from the output list (by the getWrongMessages operation). Messages with a send time greater than or equal to the receive time of the straggler message are candidate antimessages. During rollback, these messages are removed from the output list and the output list is restored to the state preceding the time the erroneous messages were sent.

The rollback recovery mechanism in time warp imposes an important precondition. In the insertEventsCheck operation's signature, the input parameter stateEvents must be of the predicate subtype defined by legalInsert?. The legalInsert? subtype ensures that there exists at least one state earlier to the time of the smallest incoming event. An informal argument for this precondition is that rollback has to restore the state preceding the start of the erroneous computations. As all the state histories are kept in the state list, a state earlier than the straggler message *must* be available. The verification of the time warp system provided in Section 5 shows that this precondition holds at any time in a time warp simulation. This ensures that the time warp protocol is able to continue until the end of simulation. However, as the simulation model consists of several simulation objects, a model of the distributed system must first be specified.

4.1.1 Distributed System

As this specification primarily targets the attributes and properties of time warp, we have specified a simple model of a distributed environment. However, the framework has been designed to support a more realistic communication model. For specifying and verifying communication-based optimizations, a more complex model is necessary. In this view of the distributed world, we define a snapshot (see Table 11) of the distributed system as a tuple containing the global communication network buffer (commManager) and the state of every process in the simulation (processes). Processes are specified as a list of simulation objects as described in the previous section. Each simulation object must have at least one state from which execution can continue as specified by the restricted type AtLeastAState?.

Simulation objects listed in processes use commManager to exchange events in a lossless, zero delay manner. Simulation objects read events addressed to them from the

TABLE 10
Simulation Object Theory

```
legalInsert?( stateEvents: StateEventTuple ): boolean =
  cons?(events(stateEvents))
  =>
  (
    lVT(smallest(stateList(state(stateEvents))))
     <
     recvTime(smallest(events(stateEvents)))
  )


insertEventsCheck(stateEvents : (legalInsert?))
  : (AtLeastAState?) =
  IF cons?(events(stateEvents)) THEN
    IF lVT(currentProcessState(state(stateEvents)))
       >=
       recvTime(smallest(events(stateEvents)))
    THEN
    state(stateEvents) with [
      inputList := signInsertList(events(stateEvents),
                                  inputList(state(stateEvents))),
      stateList := expungeAfter(
                     stateBefore(smallest(events(stateEvents)),
                     state(stateEvents)),
                     stateList(state(stateEvents))),
      currentProcessState :=
        maximum(expungeAfter(
                  stateBefore(smallest(events(stateEvents)),
                              state(stateEvents)),
                  stateList(state(stateEvents)))),
      outputList := remove(
                       getWrongMessages(
                         recvTime(smallest(events(stateEvents))),
                         outputList(state(stateEvents))),
                       outputList(state(stateEvents))
                     ),
      ether      := putAll( ether(state(stateEvents)),
                            getAntimessages(
                              getWrongMessages(
                                recvTime(smallest(events(stateEvents))),
                                outputList(state(stateEvents))
                              ))
                          )
    ]
    ELSE
    state(stateEvents) with [
      inputList := signInsertList(events(stateEvents),
                                  inputList(state(stateEvents)))
    ]
    ENDIF
  ELSE
  state(stateEvents)
  ENDIF
```

TABLE 11
Basic Simulation Theory

```
SimulationSnapshot : TYPE+ =
  [# commManager : list[(ValidEvent?)],
     processes   : list[(AtLeastAState?)] #]
```

TABLE 12
Basic Simulation Theory and Communication Theory

```
getNewOutput( processes : list[SimObjState] ) :
  RECURSIVE list[(ValidEvent?)]
cleanAllEther( processes : list[SimObjState]) :
  RECURSIVE list[SimObjState]
nextEvents(processes:list[(AtLeastAState?)]) :
  RECURSIVE list[(ValidEvent?)]

sublistProcessId(event1: (ValidEvent?), list1:list[(ValidEvent?)]) :
  RECURSIVE list[(ValidEvent?)]
```

commManager at the beginning of each simulation cycle. They output events to the commManager addressed to other simulation objects after processing has concluded. More sophisticated communications models can be implemented by altering the definitions of routines used by processes to read and write from the communications manager.

The three definitions modeling communication layers are specified in the Basic Simulation and the Communication theories (see Table 12). The observer getNewOutput scans through the processes and extracts the events from their output buffers. The list of events provided through this observer represents all the outgoing messages of all the simulation processes. After collecting all events, all local buffers are flushed. This functionality is specified by the observer cleanAllEther. Finally, messages are delivered to the receiving processes. The generation of this list of messages is dependent on the destination field and the process identifier. This is specified by the observer sublistProcessId. The process identifier is retrieved from the destination field of the event that is passed in as a parameter to sublistProcessId. One more observer on the set of processes has to be provided before the system specification can be completed. The observer nextEvents collects (into a list) from each process an event with a timestamp equal to the timestamp of the event(s) scheduled for execution. If a process is idle, no event is added to the list.

The definition of a snapshot does not enforce any restrictions on the simulation processes. But, as time warp poses several restrictions on the simulation processes, the subtype ValidSystem? is specified. The ValidSystem? specification (Table 13) is a conjunction of two properties. The first property requires that a frequent state saving scheme be applied for each process (represented by the definition of frequentState). The second property states that every simulation process must have a state earlier than GVT.

The definition of GVT used in the ValidSystem? predicate is represented by the timestamp of the smallest event in the union of commManager, getNewOutput, and nextEvents. The first part of the GVT definition is reflected by the events returned by the nextEvents operation. The second part of the GVT definition, representing all events in transit, is specified by the communication layer (commManager and getNewOutput). As a GVT value of positive infinity (simulation end) is represented by an empty list, the state property must hold only if the list is nonempty.

Having specified the snapshot of a distributed system, all that remains to be specified is transitioning between the individual snapshots. Again, the specification provides a simplified model. This model strictly states that all processes simulate and receive events exactly once between snapshots. It is important to understand that this limits the application of the model. In reality, the distributed processes might not receive any messages due to network delay or may not even execute as the system load of the distributed processor might be high. This limitation is acceptable as the network delay and other such distributed effects may be modeled by introducing predicates into the specification of the execution model. A more severe limitation is that a snapshot assumes that all processes wait between arrival of new events and execution. This assumption is only true for the initial snapshot and the final snapshot of the simulation system. Hence, it does not affect the proof of the time warp paradigm. This simplification might have to be removed if a more precise distributed system behavior is required. The distributed system behavior is specified with executeAllProcesses and allEventsToProcess.

The definition executeAllProcesses specifies the transition of processes in a snapshot during simulation execution. Each simulation process in the system receives all events from the current network state before simulation

TABLE 13
Time Warp Theory

```
ValidSystem?( snap: SimulationSnapshot ) : bool =
(
  every( (LAMBDA (process: (AtLeastAState?)): frequentState(process) ) )
  (
    (processes(snap))
  )
  AND
  (
    cons?(putAll(putAll(commManager(snap),
                        getNewOutput(processes(snap))),
                 nextEvents(processes(snap))))
    => every( (LAMBDA (process: (AtLeastAState?)):
        lVT(smallest(stateList(process)))
        < recvTime(smallest(putAll(putAll(commManager(snap),
                                getNewOutput(processes(snap))),
                                nextEvents(processes(snap))))) ) )
    (processes(snap))
  )
)
```

TABLE 14
Time Warp Theory

```
executeAllProcesses( snap: (ValidSystem?) ) : RECURSIVE
  list[(AtLeastAState?)] =
  CASES processes(snap) OF
    null: null[(AtLeastAState?)],
    cons(process, restProcesses):
      cons( simulate(
              insertEventsCheck( process, allEventsToProcess(process,
                                                    commManager(snap))
                                 )
            ),
            executeAllProcesses( (# commManager := commManager(snap),
                                    processes   := restProcesses #) )
      )
  ENDCASES
MEASURE length(processes(snap))

TimeWarpSimulation(snapshot: (ValidSystem?) )
  : (ValidSystem?) =  snapshot with [
    commManager := getNewOutput(processes(snapshot)),
    processes   :=
      executeAllProcesses(
        (# commManager := commManager(snapshot),
           processes   := cleanAllEther(processes(snapshot)) #)
      )
  ]
```

of the process is performed. In Table 14, the observer `simulate` specifies that execution is performed if events are pending. The observer `allEventsToProcess` is a rewrite rule for the communication operation `sublist-ProcessId`. The `TimeWarpSimulation` operation specifies the flow of events through the communication layer. From snapshot to snapshot, all buffers (ethers) of the simulation objects are emptied and collected in the communication network. In addition, before the execution of the simulation object, every object receives messages

TABLE 15
Lamport's Second Clock Condition

```
outputMemberNewInput : CONJECTURE
(
  FORALL (currentState: (StateAndNewInput?), event:(ValidEvent?)):
    member(event,currentInputEvents(currentState))
    =>
    recvTime(event) > stateLVT(currentState)
)

AllOutputCreatedIsInRealFuture : CONJECTURE
(
  FORALL (currentState:(StateAndNewInput?), event:(ValidEvent?)):
    member(event, processOutput(currentState)(
                    currentInputEvents(currentState),
                    currentProcessState(currentState)
                  )
          )
    =>
    recvTime(event) > newlVT(currentState)
)
```

from the network (previous snapshot) and executes with an empty `ether`.

The specification is sufficient to provide complete coverage of the time warp protocol and can be extended to show correctness of various optimizations. The framework can also be refined to model a more realistic distributed system behavior such that a proof of correctness similar to Leivent and Watro's [18] may be attempted. However, this specification is designed to be sufficient for proving correctness of optimizations to the algorithm and helps in providing a better understanding of the time warp protocol.

5 Verification of the Time-Warp Specification

The purpose of this specification is to provide a framework for better understanding the time warp paradigm and to develop correct optimizations and extensions to it. Hence, the verification of the specification concentrates on proving the assumptions and invariants stated informally by Jefferson [15] in the definition of the time warp protocol. Specifically, the following sections detail two important proofs: 1) verification of the Lamport's clock conditions and 2) verification of the invariant that each process must have one state earlier than GVT at any time since a rollback to GVT is possible. Because the PVS system provides a proof assistant that tracks and maintains proof statements in a repeatable form, only a high-level description of each proof is included here. The complete specification and the actual proofs are freely available from the authors.

5.1 Second Clock Condition for Simulation Objects

Section 3 stated the two clock conditions defined by Lamport [17] that define: 1) correct event processing and 2) correct event ordering at a node. These conditions together state the most fundamental correctness criteria for any discrete event simulator. Specifically, that individual events are not processed in negative time and that sequences of events are processed in causal order. Lamport's first clock condition is a defined property of the application and is safely stated as an invariant over events.

The second clock condition must be satisfied by the simulation protocol. The definitions `outputMemberNew Input and AllOutputCreatedIsInRealFuture` represent Lamport's second clock condition (Table 15). `outputMemberNewInput` states the clock condition for input events while `AllOutputCreatedIsInRealFu-ture` states the condition for output events. Both conditions are necessary as the simulation object must both process input events and produce output events in causal order.

The conjecture `outputMemberNewInput` (as shown in Table 15) states that any event in the set of current input events has a receive time stamp greater than the local virtual time of the current state. To prove this conjecture, a lemma is proven (not shown in this paper) showing that all events in the set of input events ready for processing have the same virtual receive time. If not, then one or more of the events must have a receive time less than others implying they should be processed first. Taking advantage of this lemma, the conjecture is proven by rewriting, instantiation of quantified variables, and case analysis.

The second conjecture in Table 15 (`AllOutputCrea-tedIsInRealFuture`) can be read as follows: Any event generated from a process during execution has a receive time greater than the local virtual time of the process before execution. To prove this conjecture, the process subtype predicate (`ValidProcess?`) is used. This combined with the attributes of the input events and the definition of LVT provide all the knowledge needed to complete this proof.

TABLE 16
Time Warp Protocol Invariant

```
TimeWarpSimulation_TCC2: OBLIGATION
(FORALL (snapshot: (ValidSystem?)):
ValidSystem?(snapshot
             WITH [commManager := getNewOutput(processes(snapshot)),
                   processes :=
                   executeAllProcesses((# commManager :=
                                           commManager(snapshot),
                                         processes :=
                                           cleanAllEther(processes(snapshot))
                                       #))]));
```

As in the previous case, the proof steps include instantiation, rewriting, and case elimination.

Note that in both conjectures the predicate subtype `ValidEvent?` is used to assure satisfaction of Lamport's first clock condition. The PVS type-check condition generator forces the verification of this property on events considered by the conjecture, automatically assuring that the property holds. TCCs are used extensively in this work to specify invariants and correctness conditions in this manner to assure their validity over changes to the specification.

5.2 The Time Warp Protocol's Invariant

While Jefferson's original definition of the time warp protocol [15] does not explicitly identify any invariants in the algorithm, it does define the invariant as a property that must be satisfied during memory management. Memory management is, however, not part of the basic time warp algorithm. The property states that, for each process, all but one saved state older than GVT can be discarded. The formal specification provides a more precise description of time warp and identifies this property as an invariant. This invariant must hold even if no global control or memory management mechanisms are specified. As with Lamport's first clock condition, PVS automatically generates type correctness conditions for this property and requires it to be proven correct whenever the specification is used.

In the specification of a simulation object, the insertion and execution of straggler events requires the existence of a state older than GVT. The need to prove this invariant arises from the fact that, by the definition of insertion, a state earlier than the smallest arriving event must be present (`legalInsert?`). As a result, the PVS system requires a proof of the following TCC generated from the definition of `TimeWarpSimulation`.

Table 16 illustrates the PVS generated proof obligation for the time warp invariant. The obligation, named `TimeWarpSimulation_TCC2`, is identified as an automatically generated type-check condition by the canonical PVS naming convention using `TCC`. This type-check condition is automatically generated because the predicate `ValidSystem?` embeds the condition of the invariant (see Section 4.1.1, Table 13). The TCC states that the execution of any legal process results in a valid global system state as defined by `ValidSystem?` property. This TCC represents an important correctness property because any `TimeWarpSimulation` system definition requires transition from one valid system state to another. Stating the correctness condition using a predicate subtype forces its verification for any specific event processing algorithm. As time warp developers use the framework to describe specific simulation systems, PVS requires maintenance of this critical invariant.

The proof of the TCC is extensive and is partitioned into several subproofs. Specifically, verification of the `ValidSystem?` predicate requires the proof of the frequent state

TABLE 17
Frequent State Saving Property

```
frequentStateAfterExecution : CONJECTURE
( FORALL ( messages: list[(ValidEvent?)],
           objects: list[(AtLeastAState?)] ):
  ValidSystem?((# commManager := messages,
                  processes   := cleanAllEther(objects)
               #))
  => every((LAMBDA (process: (AtLeastAState?)):
             frequentState(process)))
     (executeAllProcesses((# commManager := messages,
                             processes   := cleanAllEther(objects) #))) )
```

TABLE 18
Valid State Execution Property

```
ValidStateExecuteAllProcesses : CONJECTURE
( FORALL ( x: Time, messages: list[(ValidEvent?)],
           objects: list[(AtLeastAState?)] ):
  ValidSystem?( (# commManager := messages,
                   processes   := cleanAllEther(objects)
                #) )
  AND
  every( (LAMBDA (process: (AtLeastAState?)):
    lVT(smallest(stateList(process))) < x) )
  (objects)
  =>
  every( (LAMBDA (process: (AtLeastAState?)):
    lVT(smallest(stateList(process))) < x) )
  ( executeAllProcesses((# commManager := messages,
                           processes   := cleanAllEther(objects)
                           #)) ) )
```

saving property and the GVT advancement property. The following paragraphs provide a brief description of the subproofs.

Preservation of the Frequent State Saving Property: The frequent state saving property must be guaranteed at any time point in the simulation. This is stated in the theorem `frequentStateAfterExecution` (Table 17). The theorem states that, if the current system satisfies the conditions of a `ValidSystem?`, it is guaranteed that every process in the system after execution satisfies the frequent state saving property. The proof of this theorem is partitioned into two subproofs. The preservation of the frequent state property is stated for event insertion and for execution. The actual theorems (not illustrated here) are proven with the help of additional instantiations of theorems, rewrite rules, and extensive case analysis. With the help of a few subproofs, the theorem `frequentStateAfterExecution` is proven by induction on the list of simulation objects.

In some circumstances, the frequent state saving property is not applicable to a specific simulation system. For example, using an infrequent state saving scheme is one proposed optimization for time warp simulators. In such circumstances, the conjecture in Table 17 is not applicable to the proof and modification of the proof scheme is required. Chernyakhovsky et al. [6] present one such application of the framework where an infrequent state saving scheme is specified and verified.

GVT advancement: The second part of the `ValidSystem?` (Table 13) theorem states that every process maintains at least one state previous to any unprocessed event in the system. The proof of this conjecture is performed in multiple steps. The `ValidStateExecuteAllProcesses` (Table 18) conjecture establishes that if a virtual time x is greater than the time of the earliest state in a process' state list, then the virtual time x would still be greater than the earliest state in the state list of the process following execution. Induction on the list of simulation objects and the instantiation of several theorems is required to complete this proof. The theorem can not be satisfied as stated if fossil collection (removal of state history) is specified. In this case, a lower bound on the time value must be made as an additional precondition. However, the remainder of the proof remains applicable as the time value selected for fossil collection is usually the actual GVT value and the GVT value would be larger than any lower bound.

The second stage of the proof requires a transformation of the `everyListAtLeastAState` property (shown in Table 19). If it is known that every process has a state earlier than some time x, then the property is known to hold for any time y greater than x. Knowing that the system satisfies the `ValidSystem?` predicate before the execution cycle, it provides a time (GVT before execution) where the property is guaranteed. The proof of the second part can then be completed by ensuring that the new GVT is greater or equal to the old GVT. This is stated by the theorem `Time-Increment`. It should be noted that the original informal specification of the time warp protocol does not identify the valid system state property. The formal specification of the complete time warp paradigm identified the need for a definition of a valid state, that, in turn, identified a complete and correct invariant of the algorithm. By proving the properties specified by the `ValidSystem?` predicate, we are able to prove the `TimeWarpSimulation_TCC2` TCC generated by the invariant.

5.3 Verification Summary

Any verification activity must address 1) what properties were verified and 2) the sufficiency of the verification results. In this work, all assumptions and preconditions required for the correct execution of the basic time warp protocol are verified within the framework. Further, all proofs are mechanized, providing a degree of assurance in the proof process. Examples of simple preconditions and assumptions include properties such as 1) never rolling back prior to the initial state, 2) Lamport's first clock condition, and 3) properties of simulation object data

TABLE 19
State and GVT Properties

```
everyListAtLeastAState : CONJECTURE
( FORALL ( objects: list[(AtLeastAState?)], x:Time, y:Time ):
    (every((LAMBDA (process: (AtLeastAState?)):
        lVT(smallest(stateList(process))) < x ))
      (objects)
      AND
      x <= y
    )
    =>
    every((LAMBDA (process: (AtLeastAState?)):
      lVT(smallest(stateList(process))) < y ))
    (objects)
)

TimeIncrement : CONJECTURE
( FORALL ( messages: list[(ValidEvent?)], objects: list[(AtLeastAState?)] ):
  ValidSystem?((# commManager := messages,
                  processes   := cleanAllEther(objects)
                #))
  AND
  cons?(putAll(putAll(getNewOutput(objects),
                      getNewOutput
                         (executeAllProcesses((#
                             commManager := messages,
                             processes   := cleanAllEther(objects)
                                                  #)))),
               nextEvents(executeAllProcesses((#
                             commManager := messages,
                             processes   := cleanAllEther(objects)
                                                  #)))))
  => recvTime(smallest(putAll(putAll(messages,
                                     getNewOutput(objects)),
                              nextEvents(objects))))
  <= recvTime(smallest(putAll(putAll(getNewOutput(objects),
                                     getNewOutput(executeAllProcesses((#
                                       commManager := messages,
                                       processes   := cleanAllEther(objects)
                                       #)))),
                              nextEvents(executeAllProcesses((#
                                       commManager := messages,
                                       processes   := cleanAllEther(objects)
                                       #))))))
)
```

structures. Proof obligations for most of these properties are generated automatically by the PVS type-check condition generator. These verification obligations and proofs do not warrant separate discussion here; however, all details of both the specification and verification are available from the authors.

Of greater importance are the verification of Lamport's second clock condition and the time warp invariant. Lamport's second clock condition states that events are processed in causal order by simulation objects. This condition represents the most fundamental correctness criteria for any discrete event simulator and is particularly important for time warp verification. Time warp's optimistic synchronization approach allows initial out of order processing of events. By supporting rollback to previous states, time warp corrects out of order event processing when it is detected. If out of order events are not detected or rollback does not behave correctly, the simulation result may be invalid. Verification of Lamport's second clock condition assures that eventually all events are eventually

processed in order regardless of the order in which they arrive at the simulation object.

The time warp invariant proof verifies several important properties of GVT and simulation object state storage. As a part of the invariant proof, GVT advancement and frequent state saving properties are verified and provide important supporting results. The objective of the invariant proof is to demonstrate that there exists a stored state earlier that GVT. Knowing that no message or event in the system has a timestamp earlier than GVT this implies that rollback operations will always be legal because a state always exists that rollback can use to undo a processed event. This is an important result as it defines an upper limit for fossil collection processes that make time warp implementations feasible.

The two lemmas used to support the invariant proof are equally important. Verifying GVT advancement shows that GVT does not decrease. Looking at GVT as a measure of system progress, this lemma implies that the distributed simulation makes progress if individual simulation objects make progress. Nothing about out of order event processing and rollback can cause the overall simulation to go backwards beyond GVT. If LVT increases in the slowest simulation object, then GVT increases. However, if LVT never increases in one simulation object, then global progress as measured by GVT halts. It is a safe assumption that individual simulators do make temporal progress, thus assuring global simulation progress.

The frequent state savings lemma assures that system state will be saved following each event execution. Its verification assures that rollback to the state prior to execution of any individual event is always possible. As noted earlier, this condition can be too strong for some optimizations. For example, infrequent state saving optimizations do not save states following each event execution. This allows simulation objects to consume less memory at the expense of less precise rollback.

A distributed simulation is correct when it produces simulation results that are legal results from a traditional, single process simulator. Nondeterminism of simulation models and sheer complexity make verifying this property virtually impossible. However, verifying correct causal event ordering in distributed simulation is an operational description of the same property. Given that the processing of events is uniform between the monolithic simulator and distributed simulator, processing the same events in the same order will generate the same results. Thus, without knowing the specifics of event processing, verification of Lamport's second clock condition is a strong indicator of the time warp framework's correctness. In addition, verification of state maintenance and GVT properties assures the correctness of time warp's internal data structures. Specifically, verification of the time warp invariant ensures the correctness of rollback, GVT, and state maintenance functions.

6 Conclusions

The formal framework of the time warp protocol is designed to 1) specify fundamental properties of the time warp protocol, 2) verify the correctness of basic time warp properties, and 3) provide a formal framework for research into extensions to the protocol.

The specification of the basic time warp protocol presented here closely follows the original definition by Jefferson [15]. The major components of a time warp simulator are defined along with correctness criteria. In addition to Jefferson's basic specification, additional assumptions and correctness criteria are defined. Most importantly, a new time warp invariant describing correctness properties of simulation object data structures is formally defined. The close resemblance of the formal specification and the informal description of the algorithm enables the specifiers and the developers/programmers to get a much clearer, more complete, and quicker insight into the time warp protocol or parts of it. The precision required to write the formal specifications caused the specifiers to ask detailed questions of the developers. In answering these questions, the developers considered and resolved details that previously went unnoticed. Although difficult to quantify, our experience here suggests that an easy-to-follow formal specification of the time warp protocol results in a faster, more efficient software implementation [27], [28].

Each of the verification conditions identified in the specification have been verified using the PVS mechanized proof system. The most important properties verified include Lamport's second clock condition and the time warp invariant. Verifying Lamport's second clock condition assures that, upon completion of the simulation, all events have been processed in causal order. This condition is a fundamental correctness condition for any discrete event simulation system. Verifying the time warp invariant assures that rollback is always possible and establishes that time warp simulations make progress. The frequent state saves condition assures that rollback can be performed for any processed event by ensuring the prior state is available. The GVT advancement property assures that GVT never decreases and that, if all simulation objects make temporal progress, then the overall simulation makes progress.

The reusability and extensibility of the time warp framework represent the most difficult objective to quantify and evaluate. Extensibility is achieved by specifying only critical aspects of the time warp system. Domain specific properties can be added by further defining event processing, optimizations to state saving and rollback algorithms, and other generally defined functions. Verification obligations are expressed as predicate subtypes on parameters whenever possible. Thus, PVS generates obligations for, and manages the proofs of, important properties such as the time warp invariant. Most importantly, the formal verify the correctness of infrequent state saving algorithms [6], 2) represent and verify continuous-time and discrete-time synchronization protocols [9], [10], [11] in a mixed signal simulation environment, and 3) efforts are ongoing to specify and verify several distributed snapshot estimation algorithms.

From our experience with the formal specification activity, we have redesigned and rebuilt a general purpose parallel discrete-event simulator called WARPED [28]. The design of the interface classes of the WARPED simulation

kernel have been heavily influenced by the formal specification. However, it is difficult to assess the savings gained by undertaking the specification activity. Naively, it can be claimed that performing specification and verification significantly reduced implementation time for the prototype system. However, supporting such a statement with quantitative data is difficult. This is not our first experience with developing a parallel simulator [21], [27] and, thus, our previous experience contributed substantially to the improved development time. However, this was our first attempt at building a general purpose object-oriented parallel simulation kernel, so differences in the development goals did exist. Furthermore, time to working prototype is not necessarily an accurate measure of productivity as prototypes exist at widely different maturity levels. What can be said definitively is that components of the WARPED kernel were running after three months of design and implementation. A completed system [28] was available after six months. Furthermore, both the early components and implemented system were extremely stable for their relatively early development stage. Experience implementing parallel simulators suggests that this represents a significant time savings.

Limitations of the specification framework identified in the paper include an ideal communications assumption and the behavior of simulation object snapshots. Communications between time warp objects is assumed to be lossless and instantaneous. More realistic communication models must be developed if an application considers the underlying communications system. The snapshot model enforces a restriction that all processes simulate and receive events exactly once between snapshots. As stated earlier, in implemented systems, a simulation object may not receive or process events between snapshots. The snapshot model also assumes that all processes wait between the arrival of new events and execution. This assumption does not dramatically affect this verification activity. Finally, the use of list types to store events and processes introduces an ordering that is not utilized by the specification. Routines used to read from and write to the communications manager do not explicitly take advantage of this ordering. However, eliminating the ordered property by using sets to represent the events being communicated and active simulation objects would eliminate any possibility that the use of lists adversely affects the verification result.

Potential limitations of the verification performed deal primarily with the sufficiency of Lamport's second clock criteria. Although proper causal ordering of events is a strong correctness criteria, it remains to be proven that this is mathematically sufficient. Specifically, it is appealing to argue that, if events are processed in proper causal order, the distributed simulation is correct with respect to a traditional simulation of the same system. The probabilistic behavior of many simulated systems complicates verification of this argument substantially. However, causal ordering of event processing is a major goal for any parallel distributed event simulation, making it sufficient as a correctness criteria in this work.

The formal time warp specification provides a complete and verified framework of the original time warp protocol. The specification provides both a correctness proof for time warp and a platform for verifying future optimizations and extension. This paper presents the specifications of major time warp components along with verification criteria used to assure correctness. An abstract discussion of the verification activities is provided along with commentary on the rationale behind, and sufficiency of, each verification condition. Mechanisms supporting reuse and extension are discussed throughout the specification and verification descriptions. Finally, objectives of the specification activity and how those objects are met, along with limitations of the model and verification conditions are presented.

REFERENCES

[1] E.L. Acuna, J.P. Dervenis, A.J. Pagones, F.L. Yang, and R.A. Saleh, "Simulation Techniques for Mixed Analog/Digital Circuits," *IEEE J. Solid-State Circuits,* vol. 25, no. 2, pp. 353-363, Apr. 1990.

[2] B.A.A. Antao and A.J. Brodersen, "Behavioral Simulation for Analog System Design Verification," *IEEE Trans. Very Large Scale Intergration Systems,* vol. 3, no. 3, pp. 417-429, Sept. 1995.

[3] H. Bauer, "Verteilte Diskrete Simulation Komplexer Systeme," PhD thesis, Technische Univ. München, 1994.

[4] S. Bellenot, "Global Virtual Time Algorithms," *Society for Computer Simulation, Distributed Simulation,* pp. 122-127, Jan. 1990.

[5] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, Feb. 1985.

[6] V. Chernyakhovsky, P. Frey, R. Radhakrishnan, P.A. Wilsey, P. Alexander, and H.W. Carter, "A Formal Framework for Specifying and Verifying Time Warp Optimizations," *Proc. Fourth Int'l Workshop Formal Methods for Parallel Programming: Theory and Applications (FMPPTA '99),* Apr. 1999.

[7] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A Tutorial Introduction to PVS," *Workshop Industrial-Strength Formal Specification Techniques, (WIFT '95),* Apr. 1995, available with specification files, from http://www.csl.sri.com/wift-tutorial.html.

[8] J. Fleischmann and P.A. Wilsey, "Comparative Analysis of Periodic State Saving Techniques in TimeWarp Simulators," *Proc. Ninth Workshop Parallel and Distributed Simulation (PADS '95),* pp. 50-58, June 1995.

[9] P. Frey, "Protocols for Optimistic Synchronization of Mixed-Mode Simulation," PhD thesis, Univ. of Cincinnati, Aug. 1998.

[10] P. Frey, K. Nellayappan, V. Shanmugasundaram, R.S. Mayiladuthurai, C.L. Chandrashekar, and H.W. Carter, "SEAMS: Simulation Environment for VHDL-AMS," *Winter Simulation Conf. (WSC '98),* Dec. 1998.

[11] P. Frey, R. Radhakrishnan, H.W. Carter, and P. Wilsey, "Optimistic Synchronization of Mixed-Mode Simulators," *Int'l Parallel Processing Symp., (IPPS '98),* pp. 694-699, Mar./Apr. 1998.

[12] R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM,* vol. 33, no. 10, pp. 30-53, Oct. 1990.

[13] S. Ghosh, "On the Proof of Correctness of Yet Another Asynchronous Distributed Discrete Event Simulation Algorithm (YADDES)," *IEEE Trans. Systems. Man and Cybernetics–Part A: Systems and Humans,* vol. 26, no. 1, pp. 68-80, Jan. 1996.

[14] G.C. Gopalakrishnan and M. Fujimoto, "Design and Verification of the Rollback Chip Using Hop: A Case Study of Formal Methods Applied to Hardware Design," *ACM Trans. Computer Systems,* vol. 11, no. 2, pp. 109-145, May 1993.

[15] D. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 3, pp. 405-425, July 1985.

[16] B. Kannikeswaran, R. Radhakrishnan, P. Frey, P. Alexander, and P.A. Wilsey, "Formal Specification and Verification of the pGVT Algorithm," *Industrial Benefit and Advances in Formal Methods, (FME '96),* Mar. 1996.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[18] J.I. Leivent and R.J. Watro, "Mathematical Foundations for Time Warp Systems," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 5, pp. 771-794, Nov. 1993.

[19] Y.-B. Lin, "Determining the Global Progress of Parallel Simulation with FIFO Commmunication Property," *Information Processing Letters,* vol. 50, pp. 13-17, 1994.
[20] Y.-B. Lin and E. Lazowska, "Determining the Global Virtual Time in a Distributed Simulation," *Int'l Conf. Parallel Processing,* pp. 201-209, 1990.
[21] D.E. Martin, T. McBrayer, and P.A. Wilsey, *WARPED: A Time Warp Simulation Kernel for Analysis and Application Development,* 1995, available on the WWW at http://www.ece.uc.edu/~paw/warped/.
[22] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *J. Parallel and Distributed Computing,* vol. 18, no. 4, pp. 423-434, Aug. 1993.
[23] N. Shankar, S. Owre, and J.M. Rushby, *The PVS Proof Checker: A Reference Manual,* Menlo Park, Calif.: Computer Science Laboratory, SRI Int'l, Feb. 1993.
[24] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, *PVS,* System Guide, 2.3 ed. Menlo Park, Calif.: SRI Int'l Computer Science Laboratory, Sept. 1999.
[25] S. Owre, N. Shankar, and J.M. Rushby, *The PVS Specification Language,* Menlo Park, Calif.: Computer Science Laboratory, SRI Int'l, Feb. 1993.
[26] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert, *PVS* Language Reference. Menlo Park, Calif.: Computer Science Laboratory, SRI Int'l, Sept. 1999.
[27] J. Penix, D. Martin, P. Frey, R. Radhakrishnan, P. Alexander, and P.A. Wilsey, "Experiences in Verifying Parallel Simulation Algorithms," *Proc. Second Workshop Formal Methods in Software Practice (FMSP-98),* M. Ardis, ed., pp. 16-23, Mar. 1998.
[28] R. Radhakrishnan, D.E. Martin, M. Chetlur, D.M. Rao, and P.A. Wilsey, "An Object-Oriented Time Warp Simulation Kernel," *Proc. Int'l Symp. Computing in Object-Oriented Parallel Environments (ISCOPE '98),* D. Caromel, R.R. Oldehoeft, and M. Tholburn, eds., pp. 13-23, Dec. 1998.
[29] R. Rönngren and R. Ayani, "Adaptive Checkpointing in Time Warp," *Proc. Eighth Workshop Parallel and Distributed Simulation (PADS 94),* pp. 110-117, July 1994.
[30] M. Rumsey and J. Sackett, "An ASIC Methodology for Mixed Analog-Digital Simulation," *IEEE Micro 8,* pp. 34-40, Aug. 1990.
[31] J. Rushby and D.W.J. Stringer-Calvert, "A Less Elementary Tutorial for the PVS Specification and Verification System," Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI Int'l, Menlo Park, Calif., June 1995, revised, July 1996, available, with specification files, from http://www.csl.sri.com/csl-95-10.html.

Peter Frey received the Dipl.-Ing. (FH) (equivalent to the MS degree) in computer engineering from the Fachhochschule Ulm (Germany) in 1994. He continued his research at the University of Cincinnati and received the PhD degree in 1998. His research focused on distributed simulation of mixed-signal Circuits and the area of formal software engineering. In the fall of 1998, he joined Cadence Design Systems, Inc., where he centers his research interests on the design and implementation of mixed-language simulators for Spectre/Spice, Verilog-AMS, and VHDL-AMS. He continues to pursue the application of formal software engineering methods in an commercial environment. Dr. Frey is a part time lecturer at San Jose State University in San Jose, California.

Radharamanan Radhakrishnan received the BS degree in electrical and computer engineering from the University of Madras in 1993. Currently, he is a PhD student in the Department of Electrical and Computer Engineering and Computer Science at the University of Cincinnati. His current research interests include parallel discrete event driven simulation, parallel processing, formal methods, and networks.

Harold W. Carter received the BSEE degree from San Jose State University, California in 1966, the MSEE degree from the University of California, Santa Barbara in 1970, and the PhD degree in electrical engineering from the University of Southern California in 1980. For the last 15 years he has been a professor at the University of Cincinnati. Prior positions include professor at the Air Force Institute of Technology, Air Force project officer, and research engineer at the Lawrence Research National Laboratory. His research interests include parallel computation, mixed discrete/continuous simulation, and VLSI architectures. Dr. Carter is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of the ACM.

Philip A. Wilsey received the BS degree in Mathematics from Illinois State University, and the MS, and PhD degrees in computer science from the University of Louisiana at Lafayette. Currently, he is an associate professor in the Department of Electrical and Computer Engineering and Computer Science at the University of Cincinnati. His current research interests are parallel and distributed processing, parallel discrete event simulation, mixed-signal modeling and simulation, computer aided design, formal methods and design verification, and computer architecture. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Perry Alexander received the BSEE degree in 1986, the BSCS degree in 1986, the MSEE degree in 1988, and the PhD degree in 1992, all from The University of Kansas. Currently, he is an associate professor in the EECS Department and Information and Telecommunications Technology Center at The University of Kansas. From September 1992 through September 1999 he was a faculty member in the ECECS Department at The University of Cincinnati. His current research interests include systems-level design, systems design languages, formal verification and specification, and component retrieval and reuse. He is a senior member of IEEE where he chairs the Engineering of Computer-Based Systems Technical Committee, a member of the IEEE Computer Society, a member of ACM, and Sigma Xi.