A Formal Framework for Specifying and Verifying Time Warp Optimizations

Victoria Chernyakhovsky, Peter Frey, Radharamanan Radhakrishnan,
Philip A. Wilsey, Perry Alexander and Harold W. Carter

(vchernya,pfrey,ramanan,paw,alex,hcarter)@ececs.uc.edu
University of Cincinnati, PO Box 210030, Cincinnati, OH 45221–0030

Abstract. Parallel and distributed systems are representative of large and complex systems that require the application of formal methods. These systems are often unreliable because implementors design and develop these systems without a complete understanding of the problem domain; in addition, the nondeterministic nature of certain parallel and distributed systems make system validation difficult if not impossible. To address this issue, the application of formal specification and verification to a class of parallel and distributed software systems is presented in this paper. Specifically, the Prototype Verification System (PVS) is applied to the specification and verification of the Time Warp protocol, a distributed optimistic discrete event simulation algorithm. The paper discusses how the specification of the Time Warp protocol can be mechanized within a general-purpose higher-order theorem proving framework like PVS. In addition, the paper presents the extensibility of the specification to address and verify different aspects and optimizations of the basic Time Warp protocol. As an illustrative example, our experiences in specifying and verifying the infrequent state saving optimization to the basic Time Warp protocol is reported in the paper.

1 Introduction

Discrete-event simulation (DES) is a valuable design tool for several problems in engineering, computer science, economics and military applications. As many of these applications require enormous resources, parallel discrete-event simulation (PDES) is of considerable interest. Several parallel discrete-event simulation algorithms with different attributes have been reported in the literature [7]. This is due to the availability of several simulation paradigms with an assortment of optimizations. Parallel discrete-event simulation paradigms are generally classified as being *conservative* or *optimistic*. Conservative protocols [1] strictly avoid causality errors, while optimistic protocols, such as Time Warp [8] allow causality errors to occur, but implement some recovery mechanism. The Time Warp protocol is based on the *virtual time* paradigm first introduced by Jefferson [8]. Since then, the PDES research community has been actively pursuing the development of algorithmic as well as data structure based optimizations. In addition, as the popularity of the optimistic discrete-event simulation paradigm increases,

novel simulation protocols based on the original paradigm are constantly being developed.

However, the complexity and the nondeterministic execution behavior of the Time Warp protocol make it difficult to develop modifications or extensions to the basic protocol. Years of development have shown that even a simple implementation of the basic Time Warp protocol is often error prone due to difficulties in understanding the protocol. Formal specification and verification provides a solution to these problems. Formal specification eliminates ambiguities concerning the algorithm description and clearly identifies the required data structures. This results in better software designs and decreases the introduction of errors in the coding phase. Since software development is often conducted in parallel, a precise definition of the individual modules is required to avoid detection of errors late in the development phase. Formal specification provides these precise interface definitions and allows the designer to state and prove properties of individual program modules.

Despite these advantages, formal specification & verification is rarely applied on large, complex systems and is often used on smaller subsystems (modules) only. The development of module specifications limits their re-usability and a verification of the complete system is not possible. The extensible formal framework for optimistic simulation protocols [4] extendend in this paper, takes a different approach. The specification is modular and encloses the complete Time Warp simulation protocol. Modifications or extensions to the basic protocol can now be included into the framework and only the invariants of the simulation protocol will have to be re-proven. This eliminates error prone assumptions in the design of a subsystem. To illustrate the extensibility of the formal framework, the formal specification and verification of the infrequent state saving [3, 14] optimization to the basic Time Warp protocol was attempted and is reported in this paper. In addition, the framework also provides support for new optimistic simulation algorithm designs. In this case, their correctness can be verified using bisimulation [11]. The complete PVS specification of the Time Warp protocol and the specification of the infrequent state saving optimization to the Time Warp protocol are available off the WWW at http://www.ececs.uc.edu/~pfrey/pvs.

The remainder of the paper is organized as follows. Section 2 describes informally the basic Time Warp protocol and its associated components. A brief overview of the formal specification framework in presented in Section 3. In addition, the mapping of the individual components of the Time Warp protocol to their formal representation is detailed. The changes that were required to formally specify the infrequent state saving optimization is presented in Section 4. Section 5 then briefly describes the verification of the infrequent state saving specification. The conclusions, presented in Section 6, reevaluates the formal representation and the applicability of the formal framework in research and development.

2 Background

In a Time Warp synchronized discrete event simulation, *virtual time* [8] is used to model the passage of the time in the simulation. Changes in the state of the simulation occur as *events* are processed at specific virtual times. In turn, events may schedule other events at future virtual times. The virtual time defines a total order on the events of the system. The simulation state (and time) advances in discrete steps as each event is processed. The simulation is executed via several simulator processes, called simulation objects. Each simulation object is constructed from a physical process (PP), its current state, and three history queues. Figure 1 illustrates the structure of a simulation object. The input and the output queues store incoming and outgoing events respectively. The state queue stores the state history of the simulation object. Each simulation object maintains a clock that records its Local Virtual Time (LVT) and stores the LVT in its current state. Simulation objects interact with each other by exchanging time-stamped event messages. One departure from Jefferson's original definition [8] of Time Warp is that simulation objects are placed into groups called "logical processes" (LPs).

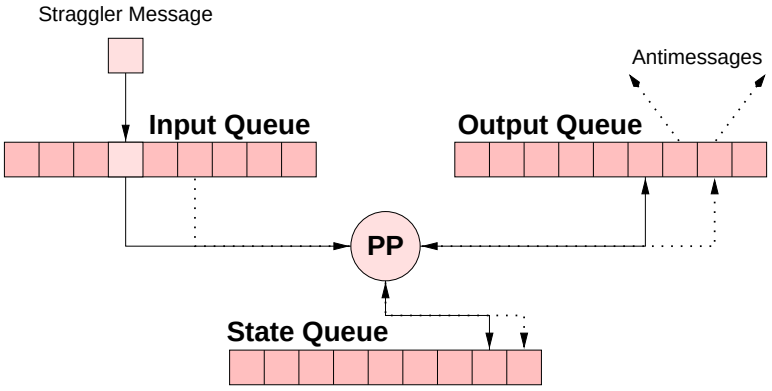

Fig. 1. A Simulation Object in a Time Warp Simulation

In addition, simulation objects in a Time Warp simulation execute their local simulation autonomously, without explicit synchronization. A causality error arises if a simulation object receives a message with a time-stamp earlier than its LVT value (a *straggler* message). In order to allow recovery, the state of the simulation object and the output events generated are saved in history queues as each event is processed. When a straggler message is detected, the erroneous computation must be undone — a *rollback* occurs. The rollback process consists of the following steps: (i) the state of the simulation object needs to be restored to a state prior to the straggler message time-stamp, and (ii) erroneously sent

output messages need to be canceled (by sending *anti-messages* to nullify the original message).

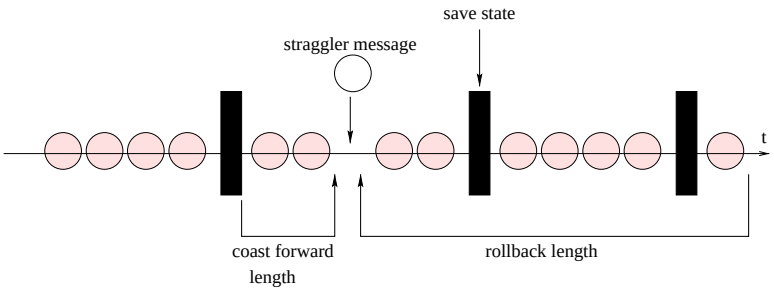

Fig. 2. Rollback and Coasting Forward ($\chi = 4$)

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 Check-pointing* [3, 14]. Figure 2 illustrates a process rolling back upon the receipt of a straggler message (white circle). The shaded circles denote events and the filled boxes denote state saving points. Since check-pointing does not occur after every processed event, a rollback will have to re-execute any intermediate events. This additional processing is called the *coast forward* phase. Thus, an optimal checkpoint interval must balance the cost of check-pointing vs. the cost of the coast forward phase. Excessively large check-pointing periods will have long coast forward phases but low check-pointing cost. The opposite is true for excessively small check-pointing periods. Using periodic state saving, the arrival of a straggler message may require the system to rollback to an earlier state than necessary and coast forward [7]. While coasting forward, no messages are sent out to the other processes in the system (the previously sent messages are correct). The difficulty of periodic state saving is determining an appropriate fixed frequency for check-pointing.

The global progress time of the simulation or Global Virtual Time (GVT), is defined as the lowest of the local virtual time (LVT) values of the simulation objects [7, 10] and the times of all messages in transit. Periodic GVT calculation is necessary to reclaim memory space used by history queues; history items with a time-stamp lower than GVT are no longer needed, and are deleted to make room for new history items.

3 The Formal Specification Framework

The specification is constructed using the Prototype Verification System (PVS). PVS provides a formal specification language, a powerful theorem prover, and a set of pre-defined basic types and axioms in the prelude file. The PVS specification language is based on higher order logic with built-in types such as booleans, integers, reals and uninterpreted (user-defined) types. To represent complex types, PVS supports functions, sets, tuples, records, enumerations, and abstract data types. In order to preserve consistency of the specification, a "conservative extension" style is provided by PVS and was extensively used in this specification. This style combines signature definition and axiomatic description in a single statement preventing the introduction of inconsistencies when new axioms are introduced. The description of the specification uses `Typewriter` font to identify names of axioms and theorems and Sansf Serif for theory names.

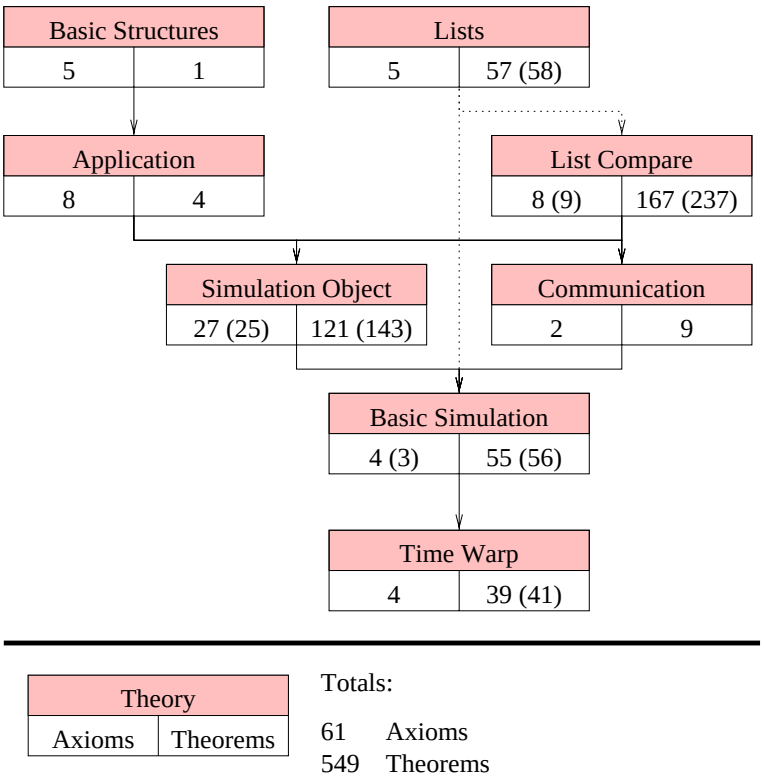

Fig. 3. Hierarchy of Theories

The specification of the infrequent state saving optimization was performed as an extension to an already existing formal specification of the Time Warp protocol [4, 6]. This formal framework follows the original definition of the Time Warp protocol by Jefferson [8] very closely. The specification provides a structured view of the protocol, identifies invariants and other Time Warp requirements (pre and post conditions). The framework is complete as it also provides a complete proof of the protocol. Figure 3 illustrates the hierarchical structure of the theories provided by the formal framework. Some of these theories were modified to include a generic proof of the infrequent state saving algorithm. The figures in the brackets represent the change in the number of the axioms or theorems as a result of the modification to incorporate the infrequent state saving specification. In addition, due to space constraints, only the modifications to the specification are detailed in this paper. For a more in depth description of the framework, please refer to [4] and [6].

Theories Lists and List Compare are parameterized theories and provide the major Time Warp list functionalities. The parameterized theories enable the list specification to be used as state and event lists (input, output) in the Time Warp structure and as message buffers in the simple communication model. The theory Basic Structures provides the generic definition of an `Event` and `State` which is annotated by the actual elements of the Application theory. The Application theory provides an abstract definition of simulation time (`Time`), process state (`StateVariables`), message content (`MsgBody`), and process identifiers (`ProcessId`). Due to this representation, the specification is applicable to any simulation model satisfying the general requirements specified in the Application theory.

The generic list structure and the representation of the application process is combined in the Simulation Object theory. This theory provides the specification of a simulation object (`SimObjState`) from the elementary Time Warp structure (see Figure 1). At any given virtual time, a state of a simulation object is described by the state of the input queue (`inputList`), the output queue (`outputList`), the state queue (`stateList`), process behavior (`process`, `processOutput`), the current state (`currentProcessState`), and the process identifier (`id`). In addition, the specification includes a local communication buffer (`ether`), which is different from the original definition of the Time Warp algorithm, but was included to model the distributed environment. The type definitions of `inputList`, `outputList`, `stateList`, `process`, `processOutput`, `currentProcessState` and `id` are all defined separately (Basic Structures and Application theories) [6] and are imported into the Simulation Object theory.

The functionality of the simulation object is described by two axioms `execute` and `insertEventsCheck` in the Simulation Object theory. These theorems describe the core of the Time Warp algorithm. As Time Warp is based on asynchronous communication, the execution is purely event-driven (*i.e.* based on the arrival of events). This fact is incorporated in the specification as follows: the `execution` axiom requires events to be processed (pending events). When events are pending for execution, the `execute` axiom describes the transition of the sim-

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

ulation object's state during execution. The specification unambiguously identifies the change of state in the `outputList`, `stateList`, `currentProcessState`, and in the `ether` of the simulation object. Events generated during execution are inserted into the `outputList` and the `ether`. The `currentProcessState` contains the newly generated state and the same state is inserted into the state history (`stateList`). As each state is inserted into the simulation object history, the frequent state saving paradigm is modeled in the original specification.

The `insertEventsCheck` axiom defines the insertion of newly arriving events into the simulation object structure. Insertion is defined on a list of arriving events and a snapshot of the simulation object. Newly arriving events are either simply inserted into the input list or in the case of a causality error, a rollback is initiated. A causality error is determined by comparing the smallest incoming event and the LVT of the current process state. The rollback changes the `inputList`, `outputList`, `stateList`, local `ether` and the `currentProcessState` as required by the Time Warp protocol. If the Time Warp algorithm uses a frequent state saving protocol, the restored state is the state with a time stamp just smaller than the time stamp of the event responsible for the rollback. As all computed states are stored in the history, forward execution can resume immediately after the rollback is performed. The Basic Simulation and Time Warp theories combine the simple distributed communication model and the simulation object model to complete the basic formal framework. The top most axiom describes the complete Time Warp system and its advancement through wall clock time. As this axiom incorporates the complete Time Warp system, a correctness proof was performed to verify that all proof requirements are satisfied. As the PVS system has a rigorous type checker, all invariants of the algorithm were automatically identified and were proven correct in the original framework [6]. [4] contains a more in depth description of all the proof obligations and the proofs that were attempted. The following section describes how the framework was adapted to incorporate a generic infrequent state saving protocol. Parts of the aforementioned theories are presented in the following section but due to the lack of space, we could not include all the theories discussed. The specification of the protocol is followed by a description of the additional theorems proven during the verification of the protocol.

4 Specification of the Infrequent State Saving Algorithm

The specification of the infrequent state saving algorithm involved modifying both the `execute` and the `insertEventsCheck` axioms. In the frequent state saving strategy, every execution cycle terminates with the storage of the current state in the `stateList`. An infrequent state saving strategy modifies this behavior by storing newly generated states infrequently. This infrequent storage of the simulation object's state is modeled by using the boolean predicate `CheckPointInterval?`. A state is saved only when this predicate evaluates to true. This is an abstract representation of an actual infrequent state saving algorithm as it does not represent any specific interval calculation scheme. This was done purposely because there are several algorithms for the checkpoint interval calculation [3] and a generic specification was required. The interval calculation (static or dynamic) is crucial for the actual performance of the simulator. However, it does not affect the correctness of the infrequent state saving paradigm.

```
% CheckPointInterval? is a predicate which is true when
% infrequent state saving is performed
CheckPointInterval? : bool

% EXECUTION OF THE SIMULATION PROCESS
execute(currentState:(StateAndNewInput?)):(AtLeastAState?) =
  currentState with [
   % no change on the inputList
   ...
   % the state is not always stored in the stateList
   stateList  :=
    IF (CheckPointInterval? ) THEN
      cons( anState with [
             lVT := newlVT(currentState),
             stateVars := process(currentState)(
                          currentInputEvents(currentState),
                          stateVars(currentProcessState(currentState))
                          )
            ],
            stateList(currentState)
           )
     ELSE
       stateList(currentState)
     ENDIF,
   % currentProcessState is advancing
   ...
   % process specification for state change and output stays the same
   % identifier stays the same
   ...
 ]
```

The `CheckPointInterval?` axiom is incorporated in the `execute` axiom. The original axiom was modified to check on the status of the `CheckPointInterval?` predicate to determine whether or not an infrequent state saving strategy is used. This is reflected in the `execute` axiom by the simple "if clause". If the `CheckPoinInterval?` axiom evaluates to true, then it is assumed that the generic checkpoint algorithm has determined the end of a checkpoint interval. In this case, the generated state after the current execution cycle is stored in the state history. Otherwise, the state history remains unchanged. Due to this infrequent checkpointing strategy, the state history may now be incomplete and rollbacks determined by the `InsertEventCheck` axiom have to execute some additional steps to determine the correct state to rollback to. The `inputList`, `outputList`, and `ether` remain unchanged. However, the `stateList` and the `currentProcessState` need to handle the incomplete state history. As shown in the `insertEventsCheck` axiom, this is done with the help of the *coast forward* algorithm.

As introduced in the informal discussion of the infrequent state saving algorithm, coast forwarding is employed to regenerate old states in order to reach the correct state required to process the straggler message in the correct causal order. During coast forwarding, the process advances from the last error free state stored in the history to the state just before the occurrence of the causality error. This behavior is stated in the `coastforward` axiom. By looking at the interface, it can be seen that during coast forwarding, only the state is changed. There are no new events generated as execution resumes from a saved state with the same inputs and stops at the point where the new events have to be processed. The purpose of this whole activity is to regenerate the required correct state.

The progress of the coast forward algorithm is controlled by the measure function which calculates the number of events in the input list between two predefined time stamps. The two time stamps are provided through the events in the `intervalLength` axiom. The use of formal methods automatically required the proof of the termination condition of the coast forward algorithm (since it was stated in a recursive manner). An abstract of this proof is provided in the verification section. The termination condition is reached when no event remains to be unprocessed between the straggler event and the time stamp of the state created during coast forwarding. The `coastforward` axiom uses a restricted type (`legalCoastforward?`) for checking the pre-conditions of the coast forward procedure. This subtype restricts the tuple containing an event and a state to a tuple where it is guaranteed that the event has a time stamp greater than the time stamp of the state. This is true for the state last stored before the straggler event and the actual straggler event. This completes the description of the changes to the original Time Warp specification that were required to specify and prove the infrequent state saving protocol. However, the original Time Warp protocol contained an axiom called `frequentState` which contains the knowledge about the state saving history. Although the frequent state saving property is no longer satisfied and is no longer provable over the

```
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 := coastforward(
   (# event        := smallest(events(stateEvents)),
      currentState := 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
```

simulation's execution, another property of the state history is still accurate and is required for the proof of correctness. The property states the basic property of event driven simulation. Every state in the state history as well as the current state is created as a result of event execution. This information is incorporated in the `EventDrivenSimulation` axiom. The `EventDrivenSimulation` axiom establishes two properties: (i) for each state in the `stateList` there exist at least one event in the `inputList` with the same timestamp, and (ii) for any `currentProcesState` of the simulation object during execution there exist an event in the `inputList` with a corresponding timestamp.

```
coastforward(stateEvent: (legalCoastforward?)): RECURSIVE State =
  IF intervalLength(
       dummyRecv(lVT(currentProcessState(currentState(stateEvent)))),
       event(stateEvent), inputList(currentState(stateEvent))) > 0
  THEN
    coastforward(
     (# event                := event(stateEvent),
        currentState         := currentState(stateEvent) with [
          currentProcessState :=
            anState with [
              lVT       := newlVT(currentState(stateEvent)),
              stateVars := process(currentState(stateEvent))(
               currentInputEvents(currentState(stateEvent)),
               stateVars(currentProcessState(currentState(stateEvent)))
       )
            ]
]
      #) )
  ELSE
    currentProcessState(currentState(stateEvent))
  ENDIF
MEASURE intervalLength(
        dummyRecv(lVT(currentProcessState(currentState(stateEvent)))),
        event(stateEvent), inputList(currentState(stateEvent)))

EventDrivenSimulation(currentState:(AtLeastAState?)) : bool =
 every( (LAMBDA (y:State): (EXISTS (x:(ValidEvent?)):
      member(x,inputList(currentState)) AND
      lVT(y) = recvTime(x))))(stateList(currentState)) AND
  (EXISTS (event: (ValidEvent?)):
    member(event, inputList(currentState)) AND
    lVT(currentProcessState(currentState)) = recvTime(event)
  )
```

5 Verification of the Infrequent State Saving Algorithm

As seen in Figure 3, several theorems were required to be added to the original specification in order to prove the correctness of the infrequent state saving algorithm in a Time Warp simulation. However, as this proof is built on top of an already existing Time Warp framework, many of the proofs did not require modification. New proofs were only required for the theorems directly affected by the above axioms and newly created type check conditions (TCCs). The coast-forward_TCC4 is just one example of a TCC that was automatically generated by the PVS type-checker. To follow the conservative extension style of specification, a recursive definition of the coast forward procedure was required. This in turn meant that PVS will automatically request a proof of the termination of the recursive definition. The actual proof involved the use of several additional theorems involving properties based on the `intervalLength`, extensive rewriting and the proof of individual cases. As the `intervalLength` is defined over the list data structure, most of the proofs concerning `intervalLength` were simply proven by rewriting and induction. For the sake of brevity, the proof steps are not shown in this paper [1].

Another important type check condition generated by the PVS type checker is the `TimeWarpSimulation_TCC2` TCC. It states that the execution of processes results in a valid global system state identifying the `ValidSystem?` axiom as being another important invariant of the Time Warp protocol. This TCC is generated from the fact that the `TimeWarpSimulation` axiom needs to transition from one valid system state to another. The proof steps involved in the verification of this invariant are not presented in this paper. It is suffice to note that several sub-proofs needed to be proven. Informally, the proof consists of two major sub-proofs. The first sub-proof verifies that simulation objects always advance forward in virtual time. The second sub-proof verifies that newly generated messages combined with any other unprocessed messages contains timestamps greater than or equal to the earliest unprocessed message timestamp before execution. 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, which in turn identified a complete and correct invariant of the algorithm.

6 Conclusion

The formal framework of the Time Warp protocol is intended to (i) improve the comprehension of the protocol, (ii) provide a structured overview with preconditions and postconditions for the elementary building blocks, and (iii) as an initial formal model for research on the protocol. The application of the formal

[1] The complete Time Warp specification and the PVS proof files are available at http://www.ececs.uc.edu/~pfrey/pvs.

```
coastforward_TCC4: OBLIGATION
 (FORALL (stateEvent: (legalCoastforward?)):
    intervalLength(
      dummyRecv(lVT(currentProcessState(currentState(stateEvent)))),
      event(stateEvent),
      inputList(currentState(stateEvent)))
      > 0
 IMPLIES
   intervalLength(
     dummyRecv(lVT(currentProcessState(currentState(stateEvent)
       WITH
       [ currentProcessState := anState
          WITH [lVT := newlVT(currentState(stateEvent)),
           stateVars := process(currentState(stateEvent))
                        (currentInputEvents(currentState(stateEvent)),
                         stateVars(currentProcessState(currentState
                         (stateEvent))))
                ]
       ]))),
     event(stateEvent),
     inputList(currentState(stateEvent)
      WITH
      [ currentProcessState := anState
         WITH [lVT := newlVT(currentState(stateEvent)),
          stateVars := process(currentState(stateEvent))
                       (currentInputEvents(currentState(stateEvent)),
                        stateVars(currentProcessState(currentState
                        (stateEvent))))]])
    )
    <
    intervalLength(
      dummyRecv(lVT(currentProcessState(currentState(stateEvent)))),
      event(stateEvent),
      inputList(currentState(stateEvent))))
```

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

model in the software development phase has shown some promising results [9, 12]. As many developers today are educated in formal methods, the specification improves the developer's comprehension and the high level overview of the simulation system. In addition, the elimination of implementation issues from the specification leaves the developer the intellectual freedom to choose suitably optimized algorithms satisfying the pre- and postconditions.

The framework is especially conducive for conducting research on the Time Warp algorithm itself. Extensions to the basic Time Warp protocol are in the form of optimizations such as infrequent state saving, rollback optimizations and event list insertion algorithms. The introduction of these optimizations to the basic Time Warp protocol is represented by the addition of the specification of these optimizations to the formal framework. As the formal framework requires several invariants to be proven, any change in the total specification (through extension) will require the invariants to be proven again. The existence of such an extensible framework reduces the specification work to just the optimization specification (as opposed to a complete specification). In addition, as the PVS environment provides methods to view existing proofs and the ability to use proven theorems, modifications to the existing framework can be proven in a shorter time. This claim is supported by the experience gained during several modifications to the basic model (*e.g.*, the effective reuse of the theorems provided by the list specifications and limited set of axioms and theorems that had to be modified to incorporate the specification of the infrequent state saving optimization).

The principal goal of this specification effort was to extend the formal framework to specify and verify in detail the Time Warp protocol and all its optimizations. For this reason, the framework was designed in a modular fashion such that other specifiers could easily add their specifications to the framework. This will allow other specifiers to quickly specify and verify the large set of optimizations to the basic Time Warp protocol in an efficient manner. In this paper, we reported our experiences in specifying and verifying one of the many Time Warp optimizations, namely the infrequent state saving optimization. We intend to continue extending the specification to specify other Time Warp optimizations. Towards this end, efforts are undergoing to specify and verify other Time Warp optimizations such as lazy cancellation [13], several GVT algorithms [2, 10], and event list management optimizations. Also the specification is currently being extended to incorporate the specification and verification of a mixed-mode simulator [4, 5] showing the applicability and extensibility of the specification to other domains.

In short, the formal specification provides a complete verified framework of the original Time Warp protocol and the infrequent state saving optimization. It is extensible in several directions, *i.e.*, to include a more accurate distributed system or be used as specification and verification framework for any Time Warp optimization or extension. In addition, the formal specification provides a description without ambiguities on an abstract level enhancing understanding of

the algorithm and the means to prove assumptions of the algorithm with the help of the PVS theorem prover.

References

1. Chandy, K. M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM 24*, 11 (Apr. 1981), 198–206.
2. D'Souza, L. M., Fan, X., and Wilsey, P. A. pGVT: An algorithm for accurate GVT estimation. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS 94)* (July 1994), Society for Computer Simulation, pp. 102–109.
3. Fleischmann, J., and Wilsey, P. A. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)* (June 1995), pp. 50–58.
4. Frey, P. *Protocols for Optimistic Synchronization of Mixed-Mode Simulation.* PhD thesis, University of Cincinnati, August 1998.
5. Frey, P., Radhakrishnan, R., Carter, H. W., and Wilsey, P. A. Optimistic synchronization of mixed-mode simulators. In *1998 International Parallel Processing Symposium, IPPS'98* (March 30 – April 3 1998), pp. 694–699.
6. Frey, P., Radhakrishnan, R., Wilsey, P. A., Alexander, P., and Carter, H. W. An extensible formal framework for the specification and verification of an optimistic simulation protocol. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS'99)* (jan 1999), Sony Electronic Publishing Services. (forthcoming).
7. Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (Oct. 1990), 30–53.
8. Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.
9. Kannikeswaran, B., Radhakrishnan, R., Frey, P., Alexander, P., and Wilsey, P. A. Formal specification and verification of the pGVT algorithm. In *FME '96: Industrial Benefit and Advances in Formal Methods* (Mar. 1996), M.-C. Gaudel and J. Woodcock, Eds., vol. 1051 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 405–424.
10. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (Aug. 1993), 423–434.
11. Milner, R. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, New York, NY, 1989.
12. Penix, J., Martin, D., Frey, P., Radhakrishnan, R., Alexander, P., and Wilsey, P. A. Experiences in verifying parallel simulation algorithms. In *Second Workshop on Formal Methods in Software Practice* (Clearwater Beach, Florida, USA, March 4-5 1998), Co-located with ISSTA98.
13. Rajan, R., and Wilsey, P. A. Dynamically switching between lazy and aggressive cancellation in a Time Warp parallel simulator. In *Proc. of the 28th Annual Simulation Symposium* (Apr. 1995), IEEE Computer Society Press, pp. 22–30.
14. Rönngren, R., and Ayani, R. Adaptive checkpointing in Time Warp. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS 94)* (July 1994), Society for Computer Simulation, pp. 110–117.