Formal Specification and Verification of the pGVT Algorithm*

Balakrishnan Kannikeswaran, Radharamanan Radhakrishnan, Peter Frey, Perry Alexander, and Philip A. Wilsey

Computer Architecture Design Laboratory, Dept of ECECS, PO Box 210030, University of Cincinnati, Cincinnati, Ohio 45221–0030, phil.wilsey@uc.edu (513) 556–4779 (voice) (513) 556-7326 (fax)

Abstract. The time warp mechanism is a technique for optimistically synchronizing Parallel and distributed Discrete Event-driven Simulators (PDES). Within this synchronization paradigm lie numerous parallel algorithms, chief among them being an estimation of the Global Virtual Time (GVT) value for fossil collection and output commit. Because the optimistic synchronization strategy allows for temporary violations of causal relations in the system being simulated, developing algorithms that correctly estimate GVT can prove extremely difficult. Testing and debugging can also prove difficult as error situations are frequently not repeatable due to varying load conditions and processing orders. Consequently, the application of formal methods to develop and analyze such algorithms are of extreme importance. This paper addresses the application of formal methods for the development of GVT estimation algorithms. More precisely, the paper presents a formal specification for and verification of one specific GVT estimation algorithm, the pGVT algorithm. The specifications are presented in the Larch Shared Language and verification completed using the Larch Proof Assistant. The ultimate goal of this work is to develop a reusable infrastructure for GVT proof development that can be used by developers of new GVT estimation algorithms.

1 Introduction

Discrete event-driven simulation is an important modeling technique used across many disciplines including, to name a few: communication networks, weather prediction, molecular motion, and economic forecasting [8]. While widely used, desires for more accurate results stimulate a need for faster simulator throughput. In response to this need, the simulation community has turned, in part, toward the potential solutions offered by parallel processing (resulting in the emergence of the subfield of Parallel Discrete Event-Driven Simulation or PDES [9]).

* Support for this work was provided in part by the Advanced Research Projects Agency, contracts F33615–93–C–1315 and F33615–93–C–1316 monitored by Wright Laboratory and contract J–FBI–93–116 monitored by the Department of Justice. The authors also wish to thank Wright Labs and ARPA for their continuing support.

Parallel solutions for discrete-event driven simulation can be broadly classified as using either (i) a central event dispatch mechanism [2, 5] or (ii) a distributed event execution mechanism [9, 20]. Modifying sequential simulators for parallel execution using central event dispatch is reasonably simple, but provides only limited speedups. Parallel simulators using distributed control are able to exploit higher degrees of parallelism, but their implementation costs can be high. This is especially true in optimistically synchronized simulators where causality relationships can be violated and then repaired [9, 13]. Decisions about global progress and the satisfaction of termination conditions in such simulations can be difficult to make. Consequently, algorithms for such decisions are frequently difficult to develop, analyze, and test.

At the University of Cincinnati, we have been studying the acceleration of digital system simulation using the time warp optimistic synchronization strategy. As part of our investigations, we have implemented a time warp simulation called WARPED and released it for public use [17, 18]. The WARPED kernel required a five month development time and over half of that time was spent in the development and test of algorithms to solve two problems, namely: decisions about the global progress of the simulation (computing a value called Global Virtual Time, or GVT), and deciding when the simulation had terminated. The difficulty experienced in developing these sections of the WARPED kernel motivated us to consider the use of formal methods for our algorithm development.

This paper presents our experiences using formal methods to develop a specific GVT estimation algorithm called pGVT. In particular, we describe our development of a formal specification and proof for the pGVT algorithm. The algorithm is specified using the Larch Shared Language [11]. This formal specification is then used to prove the correctness of the algorithm by passing the specification through the Larch Prover[2] [10] and establishing that the system GVT increases monotonically. Similar endeveours with formal specification in the distributed environment are being actively pursued by many researchers[16, 21, 23].

The reminder of the paper is organized as follows: Section 2 provides some background information on time warp, the pGVT algorithm, and the WARPED project. Section 3 presents an overview of the Larch Shared Language and the Larch Prover. Section 4 presents the formal specification of the pGVT algorithm. Section 5 contains the proof that the pGVT algorithm works correctly. Sections 6 and 7 contain some concluding remarks and a discussion of assumptions and limitations.

2 Background

2.1 Parallel Simulation and Time Warp

In distributed discrete event driven simulation, a system is generally modeled as a group of communicating entities, referred to here as *Logical Processes* (or LPs). Each LP maintains a local clock that defines the simulation time for that

[2] Please note that in this paper the abbreviation LP does not denote Larch Prover.

LP and the LPs operate as distinct discrete event simulators, exchanging event information as necessary. Synchronization between the LPs can be either *conservative* [3, 9, 20] or *optimistic* [4, 9, 13]. Under conservative synchronization, events are processed by each LP only when it can guarantee that no causality (out of order) violation will occur. In contrast, an optimistically synchronized simulation does not strictly enforce causality constraints; instead, some mechanism to recover from a causality violation is defined. Time warp is an example of an optimistically synchronized parallel simulator.

In time warp any LP with an event to process is allowed to simulate without consideration of the progress of other LPs. Since some LPs will process ahead of others at any given (real) time, simulation time is referred to as *virtual time*, and a given object's simulation time at any given moment is called its *local virtual time* (or LVT) [9, 13]. Furthermore, since each LP simulates asynchronously, it is possible for an LP to receive an event from the past — violating the causality constraints of the simulation. Such messages are referred to as *straggler* messages. In order to recover, the LP receiving the straggler message must *rollback* to an earlier simulation time and reprocess the events in their correct order. To enable rollback, each LP must maintain a history of state and event information (Figure 1). During rollback, an LP must revert to an earlier state and cancel any output events sent while it was doing (possibly) erroneous look-ahead. This cancellation is performed by sending *antimessages* to other LPs who then remove the erroneous event message from their input queue (sometimes causing rollback).

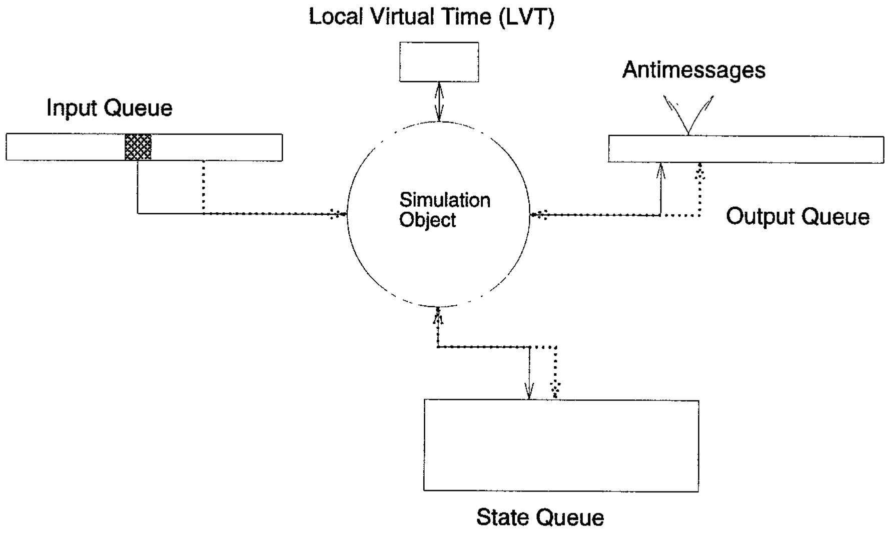

Fig. 1. A Time Warp Simulation Object

One important overhead associated with checkpointing state and event information is the memory space required for the saved data. This space can be freed only when global progress of the simulation advances beyond the (simulation) time at which the saved information is needed. The process of identifying and reclaiming this space is called *fossil collection*. The global time against which fossil collection algorithms operate is called the *global virtual time* (or GVT) and several algorithms for GVT estimation have been proposed [1, 6, 7, 15, 19, 24, 26]. In addition to its use for fossil collection, GVT is also useful for deciding when irrevocable operations (such as I/O) can be performed and, in some instances, when the simulation has completed.

2.2 The pGVT Algorithm

The pGVT algorithm [6, 7] is comprised of two functional elements: (1) the GVT management process; and (2) LGVT (local GVT) value calculation and reporting by the LPs. The GVT manager calculates, maintains, and broadcasts GVT information to LPs. Responsibility for LGVT management is distributed to each LP and is ideally organized to report new LGVT information to the central GVT manager only when failure to do so would inhibit advancement of estimates of GVT. Thus, LPs on the critical path will frequently report new LGVT information to the GVT managers, whereas LPs well in advance of the GVT will report less frequently [7]. This is described more formally below.

A single GVT manager calculates, maintains and broadcasts global GVT information to the LPs. As the LPs report their LGVT values to the GVT manager the information is saved and used to determine new estimates for GVT. When estimates of GVT increase, the GVT manager distributes (for purposes of this paper, broadcasts) this information to the LPs. Included in the GVT broadcasts, the GVT manager also computes and distributes the average rate of increase in the GVT estimates, called ΔGVT. ΔGVT is used by the LPs in determining when to report new LGVT information. More precisely, following the n^{th} GVT broadcast, the GVT manager computes:

$$GVT' = min(\{LGVT\}) \tag{1}$$

$$\Delta GVT = \frac{\sum_{i=n-k}^{n} \Delta GVT_i}{k} \tag{2}$$

where $\{LGVT\}$ is the set containing all reported LGVT values, k is the sample size used for smoothing the ΔGVT values, and ΔGVT_i is the increase in the GVT value in the i^{th} GVT broadcast. The i^{th} discrete increment in the estimated GVT value is denoted ΔGVT_i and the average of the last k increases in GVT estimates is ΔGVT. From these calculations, the GVT manager then broadcasts an ordered pair, $\langle \Delta GVT, GVT' \rangle$, to each LP.

As previously indicated, each LP independently determines when to calculate and report new LGVT information. Ideally, the LPs will report new information only when failure to do so would hinder the advancement of GVT. Thus, messages to report GVT information by each LP are reduced for all LPs except those on the critical path of GVT advancement.[3] The LPs defining the critical path frequently report the GVT information and allow an accurate estimate of GVT. More formally, the LPs report GVT information as follows:

1. Each LP calculates GVT information and saves it in a buffer called LGVT (local GVT). LGVT is the smaller of the minimum timestamp of all unprocessed events in the event queue and the minimum timestamp of all unacknowledged output messages.
2. Each LP maintains a ratio of the (real) time for a message to be sent and acknowledged to the GVT manager (denoted by t_{mesg}) and the average (real) time between successive GVT updates (denoted by ΔRT_i). That is, if K denotes the aforementioned ratio, then on the n^{th} GVT broadcast K is defined as:

$$K = \frac{t_{mesg}}{\frac{\sum_{i=n-k}^{n} \Delta RT_i}{k}} \tag{3}$$

   where k is the sample size for smoothing. K helps trigger the calculation of new LGVT information. Informally, $\lceil K \rceil$ is the number of ΔGVT cycles required to report new information to the GVT manager. Because the time required to report a value from a LP to the GVT manager may vary based on processor localities, K is computed locally.
3. Each LP recalculates and reports new LGVT information whenever:
   (a) The LP receives a straggler message with a timestamp smaller than the current LGVT value. The LP reports the new LGVT value (which will be lesser than or equal to the straggler message time) to the GVT manager before acknowledging receipt of the straggler message. When the GVT manager reports acknowledges the report, the LP acknowledges and processes the straggler message.
   (b) The broadcast GVT value approaches the current LGVT value. Reporting of new LGVT information is triggered following a just-in-time policy to ensure the most aggressive advancement of GVT. More formally, the LP computes (and reports) a new LGVT value whenever:

$$GVT + \lceil K \rceil * \Delta GVT \geq LGVT \tag{4}$$

   holds. Informally, the factor $\lceil K \rceil * \Delta GVT$ denotes the expected increase in the GVT value over the real time interval required to send (and receive acknowledgment of) a message to the GVT manager. If the GVT value plus the expected increase exceeds the last reported LGVT value, a new value for LGVT should be calculated and reported. Failure to do so would likely inhibit the advancement of GVT.

[3] A LP is said to be on the critical path if its reported LGVT value is the minimum of all values thus becoming the new GVT update.

3 An Overview of Larch

Many formal specification languages could have been used to specify the pGVT algorithm. The alternatives considered included Z [25], Larch [11] and CSP [12]. Larch was selected due to tool availability, the two-tiered specification style, and local expertise. Specifications for the pGVT algorithm components were written in the Larch Shared Language (LSL) [11] and verified using the Larch Prover [10].

The Larch style of specification is described as a two-tiered approach because specifications are written using two languages. The lower tier is written using the Larch Shared Language (LSL). LSL is an algebraic specification language that is used to model abstract data types. The functional unit of a LSL specification is the trait. The first line of the trait gives the name of the specification and declares the trait. The reminder of the specification is given in three parts: (1) the `introduces` section defining operation signatures and sorts; (2) the `asserts` section defining axioms over operations; and (3) the `implies` section defining proof obligations.

The `introduces` section specifies the operators by their signatures. A signature defines the sorts for the operator's domain and range. The `asserts` section specifies relationships among the operators using equational logic and induction rules by specifying generators for sorts. Finally, the `implies` section specifies equations that should be provable from the introduces and asserts sections. LSL supports combining specifications through parameterized inclusion much like macro expansion.[4]

A Larch specification's upper tier is an interface specification describing the specified component's interface. Interface specifications are written using a Larch Interface Language (LIL) that is tailor made to represent the target application language's calling conventions and language structures. LSL structures are referenced from LIL specifications providing common structures and behaviors in an application language independent manner. Since the algorithm being specified is written in C++, the Larch/C++ LIL is used for the GVT interface specification. The pGVT algorithm's behavior is specified using LSL making it accessible to any LIL, not simply Larch/C++. As this work deals with the pGVT's correctness independent of the specific Larch/C++ implementation, the interface specifications are not presented here.

The Larch Prover [10] is a proof assistant compatible with the LSL specifications. The equations specified in an LSL asserts section are converted into rewrite rules, deduction rules and induction rules. The Larch Prover allows the user to make conjectures and applies the rules using either forward or backward inferencing techniques. The Larch Prover's primary uses include checking for consistency and theory containment.

[4] LSL contains a number of additional constructs not used in this specification.

4 The Specification of the pGVT Algorithm

The formal specification of the model consists of two parts defining the GVT manager and the simulation model from a logical process view point. This two part approach supports direct representation of the pGVT algorithm's two functional elements described in Section 2.2. The GVT Manager trait specifies GVT calculation and LGVT value update. The trait provides two operations representing these activities for use in the LP specification. The Model trait specifies the behavior of a logical process. This logical process can be any one of the LPs present in the system. The LP model processes messages and interacts with the GVT manager to update its LGVT entry and obtain new values of GVT.

4.1 The GVT Manager

The GVT manager trait defines a GVT manager's behavior. The GVT manager maintains a record of LGVT values for each LP. The interface to the GVT manager consists of function to: (a) update an LP's associated LGVT value and (b) calculate and return GVT.

```
Manager : trait

includes FiniteMap(LPRecords, LPName, LatestLGVT)
includes Natural(LatestLGVT)

introduces

        Mgr             : LPRecords → GVTmgr
        sendGVT         : GVTmgr, LPName, LatestLGVT → GVTmgr
        gvt             : GVTmgr → LatestLGVT

        empty           : → LPInfo
        range           : LatestLGVT, LPInfo → LPInfo
        getrange        : LPRecords → LPInfo
        min_in_range    : LPInfo → LatestLGVT

        min             : LPRecords → LatestLGVT

asserts
  LPInfo generated by empty, getrange
  ∀ LastReportLGVT : LPRecords,
          lgvt1, lgvt2 : LatestLGVT,
          lpname : LPName,
          lpinfo : LPInfo

  gvt(Mgr(LastReportLGVT)) == min(LastReportLGVT);
  sendGVT( Mgr(LastReportLGVT), lpname, lgvt1) ==
      Mgr(update(LastReportLGVT,lpname,lgvt1));
  getrange({}) = empty;
  getrange(update(LastReportLGVT,lpname,lgvt1)) ==
      range(lgvt1,getrange(LastReportLGVT));
  min(LastReportLGVT) == if getrange(LastReportLGVT) ¬= empty
    then
```

```
      min_in_range(getrange(LastReportLGVT))
    else
      0;
  min_in_range(range(lgvt1,lpinfo)) ==
    if lpinfo = empty then
      lgvt1
    else
      if lgvt1 < min_in_range(lpinfo) then
        lgvt1
      else
        min_in_range(lpinfo);
  update(update(LastReportLGVT,lpname,lgvt2),lpname,lgvt1)
     = update(LastReportLGVT,lpname,lgvt1);
  getrange(update(LastReportLGVT,lpname,lgvt1)) ¬= empty
implies
  ∀ LastReportLGVT : LPRecords, lgvt1 : LatestLGVT, lpname : LPName
  lgvt1 ≥ min(LastReportLGVT) ⇒
    gvt(sendGVT(Mgr(LastReportLGVT), lpname, lgvt1))
      ≥  gvt(Mgr(LastReportLGVT) );
```

Manager Trait Operators A GVT manager is represented by sort *GVTmgr* generated by the operator Mgr. Its principle data structure is a finite map from each LP name to that LP's most recently reported LGVT. This finite map is updated when a new LGVT is reported by any LP.

Two operators are specified to update the LGVT map and obtain a new GVT value from a GVT manager. The sendGVT operator updates the value of an LGVT value stored in the GVT manager. The operator's domain is a 3-tuple, $\langle GVTmgr, LPName, LatestLGVT\rangle$, where *GVTmgr* is the current state of the GVT manager, *LPName* represents the logical process in question and *LatestLGVT* represents the new LGVT. The gvt operator is used to get the current GVT value from a GVT manager. GVT is obtained by applying the operator gvt to the GVT manager (*GVTmgr*). These two operators define the external interface of a GVT manager allowing updates to stored LGVT values and GVT value retrieval.

GVT Manager Axioms Updating an LGVT value in the GVT manager is represented as a finite map update for the LP in question. The update function is defined in the finite map trait and simply referenced in the single sendGVT axiom:

```
sendGVT(Mgr(LastReportLGVT),lpname,lgvt1) ==
                    Mgr(update(LastReportLGVT,lpname,lgvt1));
```

The system GVT is simply the minimum LGVT in the finite map maintained by the GVT manager. Thus, gvt is defined:

```
gvt(Mgr(LastReportLGVT)) == min(LastReportLGVT);
```

The minimum value in the finite map is found by obtaining its range and searching it for the minimum value.

```
min(LastReportLGVT) == if getrange(LastReportLGVT) ¬= empty then
  min_in_range(getrange(LastReportLGVT))
else
  0;
```

The check for an empty range exists because it is possible to have no LPs, leading to an empty map with an empty range. In the case of an empty map, GVT will not change and remains equal to zero.

The operator min_in_range is recursively defined as follows:

```
min_in_range(range(lgvt1,lpinfo)) ==
  if lpinfo = empty then
    lgvt1
  else
    if lgvt1 < min_in_range(lpinfo) then
      lgvt1
    else
      min_in_range(lpinfo);
```

This simple specification does a linear search for the minimum value in *lpinfo*.

The operator getrange maps the domain of a finite map to a range set. An empty map results in an empty range.

```
getrange({}) = empty;
getrange(update(LastReportLGVT,lpname,lgvt1)) ==
  range(lgvt1,getrange(LastReportLGVT));
```

getrange builds the range set recursively stepping through the finite map entries. If the finite map is not empty, the range is the value of the mapping function (the map entry, lgvt1) and the range of the rest of the finite map.

The following two domain axioms are added to support the Larch Proof assistant.

1. If a range value in the finite map is updated, it can be replaced by only the new value.

   ```
   update(update(LastReportLGVT,lpname,lgvt2),lpname,lgvt1) =
     update(LastReportLGVT,lpname,lgvt1);
   ```

2. A finite map with at least one entry cannot be empty.

   ```
   getrange(update(LastReportLGVT,lpname,lgvt1)) ¬= empty
   ```

4.2 Modeling the LPs

The logical process model defines a logical process's behavior. Each logical process communicates with the GVT manager by: (a) sending a new LGVT value; or (b) receiving a new GVT value. Thus, the interaction between an LP and the manager is completely defined by the two interface operators defined in the Manager trait. Using the Manager trait is a simple matter of including the trait in the LP specification.

```
Model : trait
includes Manager

introduces
  LP : LatestLGVT, GVTmgr → LPName
  check: LPName → LPName
  work: LPName → LPName
  progress: SystemState → SystemState
  state: SystemState, LPName → SystemState
  start : → SystemState
  straggler : SystemState → Bool
  stragglertime : SystemState → LatestLGVT
  rollback : SystemState, LatestLGVT, GVTmgr → SystemState
  MGR : SystemState → GVTmgr
  time: SystemState → LatestLGVT
  delta :→ LatestLGVT

asserts
  SystemState generated by state, start

  ∀ lpname : LPName,
     lgvt1, lgvt2 : LatestLGVT,
     mgr,curmgr : GVTmgr,
     LastReportLGVT : LPRecords,
     systemstate : SystemState

   progress(start) == state(start, LP(gvt(mgr), mgr));
   progress(state(systemstate,lpname)) ==
     if straggler(state(systemstate,lpname)) then
       rollback(state(systemstate,lpname),
        stragglertime(state(systemstate,lpname)),
        MGR(state(systemstate,lpname)))
     else
       state(state(systemstate,lpname),check(work(lpname)));

   stragglertime(state(systemstate,LP(lgvt1,mgr))) < lgvt1;
   stragglertime(systemstate) ≥ gvt(MGR(systemstate));

   rollback(state(systemstate,LP(lgvt1,mgr)),lgvt2,curmgr) ==
     if lgvt1 > lgvt2 then
       rollback(systemstate,lgvt2, curmgr)
     else
       state(systemstate,LP(lgvt1,
             sendGVT(curmgr,LP(lgvt1,mgr), lgvt1)));
```

```
    work(LP(lgvt1,mgr)) == LP(succ(lgvt1),mgr);
    check(LP(lgvt1, mgr)) ==
      if (gvt(mgr)+delta) > lgvt1 then
        LP(lgvt1, sendGVT(mgr,LP(lgvt1,mgr),lgvt1))
      else
        LP(lgvt1, mgr);

    MGR(state(systemstate, LP(lgvt1, mgr))) = mgr;
    time(state(systemstate,LP(lgvt1, mgr))) = lgvt1;

    gvt(MGR(start)) = time(start);
    time(progress(start)) ≤ time(state(systemstate,lpname));
    gvt(MGR(start)) ≤  gvt(MGR(progress(start)));
implies
  ∀ systemstate1,systemstate2 : SystemState
    time(systemstate1) ≥ gvt(MGR(systemstate1))
    progress(systemstate1) = systemstate2 ⇒
      gvt(MGR(systemstate1)) ≤ gvt(MGR(systemstate2))
```

The LP Model Trait Operators The basic model construct is the logical process. It is obtained using the operator LP with range *LPName* and domain $\langle \mathit{LatestLGVT}, \mathit{GVTmgr} \rangle$, where *LatestLGVT* represents the current LGVT value of the logical process and *GVTmgr* represents the GVT manager.

To model the progress of the GVT algorithm, modeling system state and change of state is necessary. The sort state consists of the tuple $\langle \mathit{SystemState}, \mathit{LPName} \rangle$ and stores the history of a simulation. To allow a proof by induction on the states, the initial state, start, is defined. The operator progress allows reference to the history of simulation states.

The operator check is a mapping from one logical process to another. It checks the conditions for LGVT update and changes the GVT manager's value if necessary.

In order to specify logical process execution, the operator work is defined. It maps one logical process to another and abstractly represents the change in LP state resulting from execution. The specifics of the state change are immaterial to this verification effort — only that the state changes need be represented. The only assumption made is that work increases the LPs LGVT value representing "positive" work.

Additional operators are introduced to specify the special simulation properties. A straggler operator is a mapping from a system state to a boolean type. The boolean variable indicates whether a straggler message has been processed. The operator stragglertime accesses the straggler message's arrival time. This time is used in the rollback operator, along with the present state and a reference to the GVT manager, to model rollback to an earlier state. Rollback is the inverse of work. The state change it produces always causes LGVT to decrease.

Two operators are defined to access the fields of a logical process state. MGR relates the GVT manager with its current state. The operator time references the LGVT of the state.

Finally, the operator delta represents passage of a small simulation time interval that represents the value calculated by Equation 3.

Model Trait Axioms The system is represented by states described by the state sort. To prove characteristics over system progress, proof by induction over states is necessary. The following axiom specifies the initial state:

```
progress(start) == state(start, LP(gvt(mgr), mgr));
```

The initial condition occurs when the logical process's LGVT is equal to the GVT held by the GVT manager. A change in state (progress) is modeled as shown below:

```
progress(state(systemstate,lpname)) ==
  if straggler(state(systemstate,lpname)) then
    rollback(state(systemstate,lpname),
     stragglertime(state(systemstate,lpname)),
     MGR(state(systemstate,lpname)))
  else
    state(state(systemstate,lpname),check(work(lpname)));
```

Note that the LP changes state in two ways. Either a straggler message arrives and the state is rolled back, or a logical process proceeds with its task.

The operator straggler is used in the progress operator definition. The straggler operator is an abstraction used to denote the fact that, the LP under consideration could have received this straggler message from any other LP in the system. Thus the whole system is modelled, by just focussing on any LP. If a straggler occurs, the following two statements determine the rollback time.

```
stragglertime(state(systemstate,LP(lgvt1,mgr))) < lgvt1;
stragglertime(systemstate) ≥ gvt(MGR(systemstate));
```

If a straggler message arrives, the straggler's event time is less than the LP's LGVT. In addition the event time must be greater than or equal to the GVT. Because all states up to the state whose LGVT is equal to the GVT are stored, rollback to a previous state is possible. This characteristic is asserted in the following axiom:

```
rollback(state(systemstate,LP(lgvt1,mgr)),lgvt2,curmgr) ==
  if lgvt1 > lgvt2 then
    rollback(systemstate,lgvt2, curmgr)
  else
    state(systemstate,LP(lgvt1,sendGVT(curmgr,LP(lgvt1,mgr), lgvt1)));
```

Although this appears to be an extremely strong assertion, it simply states that if a straggler arrives and GVT is managed correctly, stored state information can reconstruct the state when the straggler message should have arrived.

As long as the time of the previous state is greater than straggler time, rollback continues. When the state time is less than the straggler time, processing can continue as a state prior to message arrival has been reconstructed. This state's time is the rollback time. As mentioned in the algorithm 2.2, the rollback time must be reported to the GVT manager as the LP's current LGVT.

The following axioms model the internal behavior of a logical process.

```
work(LP(lgvt1,mgr)) == LP(succ(lgvt1),mgr);
```

The operator work describes forward progress from the logical process's point of view. When a logical process has done work, it increases its LGVT. The operator check, however has to determine whether the LGVT has reached the boundary described in Equation 4.If the condition is true, the logical process must report its LGVT to the GVT manager. Otherwise the logical process simply advances:

```
check(LP(lgvt1, mgr)) ==
  if (gvt(mgr)+delta) > lgvt1
    then LP(lgvt1, sendGVT(mgr,LP(lgvt1,mgr),lgvt1))
    else LP(lgvt1, mgr);
```

As described before, two accessor operators are defined. MGR accesses the GVT manager for the current state and time accesses the current state's LGVT.

```
MGR(state(systemstate, LP(lgvt1, mgr))) = mgr;
time(state(systemstate, LP(lgvt1, mgr))) = lgvt1;
```

The following auxiliary axioms state basic truths necessary for the proof process. They define characteristics of time and gvt in the initial state. Specifically, time and gvt in the initial state represent minimum values for each. These assertions are necessary for proof by induction.

1. The system's initial condition asserts that GVT and LGVT are initially the same for all logical processes.

   ```
   gvt(MGR(start)) = time(start);
   ```

2. A state's time will never be less than the initial state's time:

   ```
   time(progress(start)) ≤ time(state(systemstate,lpname));
   ```

3. GVT is never less than GVT in the initial state:

   ```
   gvt(MGR(start)) ≤  gvt(MGR(progress(start)));
   ```

5 The Correctness Proof

The pGVT algorithm is operating correctly when GVT monotonically increases. Each increase in GVT value represents movement forward in simulation time and thus progress towards a completed simulation. Although individual LPs may rollback, GVT should never decrease as it represents a lower floor for LGVT values. Allowing GVT to decrease causes the entire system to rollback eliminating the guarantee that "progress" is being made. Furthermore, since GVT is used for fossil collection, allowing GVT to decrease violates the assumption that throwing away information from states earlier than GVT is desirable.

The proof for this obligation is decomposed into 3 subproofs: (1) prove that GVT always increases and is calculated correctly if new LGVT values exceed GVT; (2) prove that in any state, the LGVT is always higher than the last GVT; and (3) using results from (1) and (2), show that GVT increases monotonically. Formally, the three proof obligations become:

```
(1) lgvt1 ≥ min(LastReportLGVT) ⇒
      gvt(sendGVT(Mgr(LastReportLGVT), lpname, lgvt1))
         ≥ gvt(Mgr(LastReportLGVT));
(2) time(systemstate1) ≥ gvt(MGR(systemstate1));
(3) progress(systemstate1) = systemstate2 ⇒
      gvt(MGR(systemstate1)) ≤ gvt(MGR(systemstate2));
```

5.1 Proving Conditional Monotonic GVT Increase

The primary focus of this proof is showing that the GVT manager must either make progress or maintain GVT. This represents the overall proof obligation — unfortunately it cannot be proven directly. What is shown initially is that the GVT manager definition guarantees that GVT monotonically increases given that all new LGVT values are greater than GVT. This obligation is modeled by the following Larch proposition:

```
lgvt1 ≥ min(LastReportLGVT)  ⇒
  gvt(sendGVT(Mgr(LastReportLGVT), lpname, lgvt1))
     ≥  gvt(Mgr(LastReportLGVT));
```

The proof obligation states simply that if all reported LGVT values increase, then GVT also increases or stays the same.

The prover initially tries to prove the propositions using existing axioms, formulas and rewrite rules in combination with the default proof methods. In this case, the prover does not have enough knowledge to select a more powerful proof method. Thus, the following directives are supplied by the user to the Larch Proof Assistant:

1. To reduce the complexity of the current conjecture, the formula

   ```
   Manager.5: getrange(update(LastReportLGVT, lpname, lgvt1))
                 = range(lgvt1, getrange(LastReportLGVT))
   ```

   was applied to the conjecture. The command issued by the user is

   ```
   rewrite conjecture with Manager.5
   ```

2. After analyzing the conjecture, two range cases were found – either the number of logical processes is equal to 0 or greater than 0. Hence the following command was used to continue using a proof by cases:

```
resume by cases getrange(LastReportLGVT) = empty
```

The prover continues by attempting to prove a conjecture representing each case. A third obligation, proving the two cases cover all possibilities, is discharged automatically by the prover.

3. Next, the proof continues by implication using the following LP command:

```
resume by ⇒
```

4. Finally another proof by cases is attempted. This time the proof is generated for the case when *lgvt*1 is less than the minimum value. The subproof is generated by the statement:

```
resume by cases
  lgvt1c ⊖ min_in_range(getrange(LastReportLGVTc)) = 0
  ∧  ¬(lgvt1c = min_in_range(getrange(LastReportLGVTc)))
```

The final command causes the proof to complete and the original proposition becomes a theorem for use in later proofs. Thus, we know that GVT always increases if new LGVT values increase.

Larch Prover instructions for the complete proof have the following form:

```
% Conjecture:
%  lgvt1 ≥ min(LastReportLGVT) ⇒
%    gvt(sendGVT(Mgr(LastReportLGVT), lpname, lgvt1))
%       ≥  gvt(Mgr(LastReportLGVT) );
% Proof steps:
   rewrite con with Manager.5
     % (get rid of update
     %  transform getrange into the range )
   resume by cases
     % (Either LPs exist or they don't)
     %  getrange(LastReportLGVT) = empty)
   resume by ⇒
     % (Assuming that the input is correct then the result
     %  is also)
   resume by cases
     % (for send time
     %  lgvt1c ⊖ min_in_range(getrange(LastReportLGVTc)) = 0
     %  ∧  ¬(lgvt1c = min_in_range(getrange(LastRepcrtLGVTc)))
```

5.2 Proving LGVT Less Than GVT

The second subproof conjecture is that the LGVT of any state will always be greater than or equal to the present GVT. The Larch statement representing this hypothesis is:

```
time(systemstate1) ≥ gvt(MGR(systemstate1))
```

After the transformation into the Larch Prover, the following steps guide the Larch Prover:

1. Because the statement has to hold for all possible states of the logical process, a proof by induction over states is required. The Larch Prover command to attempt a proof by induction is as follows:

   ```
   resume by induction on systemstate1
   ```

2. A close examination of the new conjecture shows that the present set of rewrite rules should be enough to prove the conjecture. In order to repeatedly apply the rewrite rules, the critical-pairs command is used to apply Knuth-Bendix [14] completion over a subset of assertions:

   ```
   critical-pairs Model* with Model*
   ```

The result is that for all LP states, LGVT is greater than or equal to GVT. Using this result and properties of work and rollback, it will be shown that LGVT associated with every next state is also greater than GVT. Thus, successive GVT values are greater than or equal to old GVT values.

5.3 Proving GVT Progress

Progress from one state to another implies that the GVT of the new state must be greater than or equal to the GVT of the old state. Because progress is defined on the state of a logical process, logical processes that make progress guarantee the growth of the GVT. This is represented by the following Larch conjecture:

```
progress(systemstate1) = systemstate2 ⇒
  gvt(MGR(systemstate1)) ≤ gvt(MGR(systemstate2))
```

Two occurrences cause an LP to change state: (1) normal forward processing (or work); or (2) processing straggler messages (or rollback). Recall that the definition of progress indicates when work or rollback applies. In either case, progress generates a new state. Using a proof by cases, it is shown that regardless of how the state changes, GVT will either change positively or not at all. The steps to aid the Larch Proof Assistant are:

1. Stating the proof obligation over state transitions allows proof by induction over states. This is written as follows:

   ```
   resume by induction on systemstate1
   ```

2. A proof by cases is attempted to show that in any state the logical process can either move forward (work) or receive a straggler message (rollback). To achieve this proof, LP is given the following command:

```
resume by cases straggler(state(systemstatelc, lpname))
```

3. Careful evaluation of the conjecture reveals that existing rewrite rules can prove the conjecture directly. However the `critical-pairs` command needs to be applied twice, because it halts after a first partial proof.

```
critical-pairs Model* with Model*
```

Larch Prover code for the final two proof obligations has the following form:

```
% Conjecture time(systemstate1) ≥ gvt(MGR(systemstate1))
% Steps :
   resume by induction on systemstate1
   critical-pairs Model* with Model*
   qed

% After stating the above axiom as a theorem we prove the following.
% Conjecture: progress(systemstate1) = systemstate2 ⇒
%                 gvt(MGR(systemstate1)) ≤ gvt(MGR(systemstate2))
% Steps :
   resume by induction on systemstate1
   resume by cases straggler(state(systemstatelc, lpstate))
   critical-pairs Model* with Model*
   critical-pairs Model* with Model*
   qed
```

This completes the proof of the desired conjecture. Specifically that GVT monotonically increases over a set of simulation states. Because GVT monotonically increases and represents a floor for LGVT, discarding LP state information prior to GVT is a legitimate and correct fossil collection algorithm.

6 Limitations and Assumptions

6.1 Perfect Communication

This model specifies a pGVT algorithm under the assumption that message passing between the GVT manager and the logical processes is a single, perfect message. The actual algorithm specifies a handshaking protocol implemented using acknowledgment messages. The current specification does not model the acknowledgment process. The assumption is made that when a logical process or the GVT manager sends a message to another logical process or GVT manager the sender waits until an acknowledgment is received.

6.2 Simulation Progress

It should be noted that successive values of GVT are guaranteed to be greater than or equal to the previous GVT value. There is no guarantee that GVT ever increases. It is possible for a simulation to maintain a single GVT value indefinitely. Showing that GVT always increases is a desirable result, but it is impossible without knowing the internal details of the LPs. It can be concluded that if all LPs' LGVT values increase, then GVT will increase. Thus, if LPs make progress, the overall system will make progress.

7 Conclusions

Parallel processing promises to deliver performance improvement needed for large modeling efforts using discrete event simulation. However, this promise requires solutions of distributed synchronization that can be difficult to deliver. This is especially true with optimistic synchronization techniques where algorithm developers must deal with non-repeatable behaviors and transient error situations. In such environments traditional testing and path analysis fails and the designer must turn to other methods for algorithm design and analysis. One possible solution is the application of formal methods for specification and proof. From this context, we have pursued the application of formal methods for two important subproblems of a time warp simulator. The first addressed the problem of demonstrating correct LP behavior and event commitment [22]. The second, described herein, addresses the problem of global time management — specifically the estimation of GVT advancement with the pGVT [7] algorithm.

In this paper, a formal specification for the pGVT algorithm has been presented with an automated formal proof that GVT increases monotonically. Although the specifications and proof are important results in themselves, this activity represents a pragmatic application of formal verification to a realistic algorithm. The pGVT algorithm is a commonly used parallel simulation algorithm and represents much more than a "toy" analysis problem. The specifications were written and verified by formal methods practitioners working with experts in parallel simulation.

One gain of this effort was the knowledge that a formal specification of an algorithm is not as daunting as it seems and as an end result, the specifiers and readers of the proof have a more precise understanding of the algorithm. The result of this exercise ultimately led to a clean interface and well structured implementation of the algorithm in the WARPED kernel [18].

Lastly, the chief final objective of this effort is the development for a proof infrastructure to support reasoning about various alternative algorithms for GVT estimation. More precisely, we hope to develop a framework for GVT proof that will support and simplify the work required by future designers of GVT algorithms.

References

1. Bauer, H., and Sporrer, C. Distributed logic simulation and an approach to asynchronous GVT-calculation. In *6th Workshop on Parallel and Distributed Simulation* (January 1992), Society for Computer Simulation, pp. 205–208.

2. Blank, T. A survey of hardware accelerators used in computer-aided design. *IEEE Design and Test of Computers 1*, 4 (August 1984), 21–39.
3. Chandy, K. M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM 24*, 11 (April 1981), 198–206.
4. Chandy, K. M., and Sherman, R. Space-time and simulation. In *Distributed Simulation* (1989), Society for Computer Simulation, pp. 53–57.
5. Denneau, M., Kronstadt, E., and Pfister, G. Design and implementation of a software simulation engine. *Computer-Aided Design 15*, 3 (May 1983), 123–130.
6. D'Souza, L. M. Global virtual time estimation algorithms in optimistically synchronized distributed discrete event driven simulation. Master's thesis, University of Cincinnati, Cincinnati, Ohio, May 1994.
7. 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.
8. Fishwick, P. A. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, Englewood Cliffs, NJ, 1995.
9. Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (October 1990), 30–53.
10. Garland, S. J., and Guttag, J. V. A guide to LP, the Larch Prover. Tech. rep., TR 82, DEC/SRC, December 1991.
11. Guttag, J. V., and Horning, J. J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.
12. Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.
13. Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.
14. Knuth, D. E., and Bendix, P. B. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. Leech, Ed. Pergamon Press, 1970.
15. Lin, Y.-B., and Lazowska, E. Determining the global virtual time in a distributed simulation. In *1990 International Conference on Parallel Processing* (1990), pp. III–201–III–209.
16. Lincoln, P., and Rushby, J. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In *Computer-Aided Verification, CAV'93* (June/July 1993), C. Courcoubetis, Ed., vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 292–304.
17. Martin, D. E., McBrayer, T., and Wilsey, P. A. warped: A time warp simulation kernel for analysis and application development, 1995. (available on the www at `http://www.ece.uc.edu/~paw/warped/`).
18. Martin, D. E., McBrayer, T. J., and Wilsey, P. A. warped: A time warp simulation kernel for analysis and application development. In *29th Hawaii International Conference on System Sciences (HICSS-29)* (January 1996). (forthcoming).
19. Mattern, F. Effecient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (August 1993), 423–434.
20. Misra, J. Distributed discrete-event simulation. *Computing Surveys 18*, 1 (March 1986), 39–65.

21. OWRE, S., RUSHBY, J., SHANKAR, N., AND VON HENKE, F. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering 27(2)* (February 1995), 107–125.
22. PENIX, J., ALEXANDER, P., MARTIN, D., AND WILSEY, P. A. Formal specification and partial verification of LVT in a time warp simulation kernel, 1995.
23. RUSHBY, J. A formally verified algorithm for clock synchronization under a hybrid fault model. *13th ACM Symposium on Principles of Distributed Computing(PODC'94)* (August 1994), 304–313.
24. SAMADI, B. *Distributed Simulation, Algorithms and Performance Analysis.* PhD thesis, Computer Science Department, University of California, Los Angeles, CA, 1985.
25. SPIVEY, J. M. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, Cambridge, 1988.
26. TOMLINSON, A. I., AND GARG, V. K. An algorithm for minimally latent global virtual time. In *Proc of the 7th Workshop on Parallel and Distributed Simulation (PADS)* (July 1993), Society for Computer Simulation, pp. 35–42.