Distributed Event-driven Simulation- Scheduling Strategies and Resource Management

Overeinder, B.J.

Citation for published version (APA):

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

The APSIS Time Warp Kernel

Main Entry: ap-sis
1: the point in an astronomical orbit at which the distance of the body
from the center of attraction is either greatest or least
2: APSE 2

—Merriam-Webster Dictionary

3.1 Introduction

The development of Parallel Discrete Event Simulation (PDES) applications is a complex design and implementation activity. Besides the complexity of parallel program development, the simulation modeler also has to think about the intrinsics of the PDES method, e.g., conservative versus optimistic synchronization and the consequent design and implementation details. For example, with conservative simulators the developer must be familiar with the issues of lookahead and known Logical Process (LP) connectivity, and with optimistic simulators he must be familiar with the issues of state saving and rollback. The lack of versatile parallel simulation environments or languages and performance analysis tools to simplify the development of PDES simulations, has hampered the acceptance of PDES in the simulation community.

The Amsterdam Parallel Simulation System (APSIS) addresses some of the problems identified above, by providing a platform that supports for experimental development of optimistic simulation protocols and applications, and the subsequent performance analysis. Specifically, requirements for computational science applications are taken into consideration to assess the potential of PDES methods to solve problems originating from, e.g., physics, chemistry, or biology. These requirements put special constraints on the design of the simulation environment and necessitate new extensions to the basic Time Warp method. Performance analysis should be an integrated part of PDES application development. At any instant, the PDES protocol or application developer must be able to validate the efficiency of his parallel design and identify performance bottlenecks. In Chapter 4, the design and implementation of the performance analysis tool is presented, including its integration with the APSIS
simulation environment, which is described in this chapter.

Section 3.2 presents PDES languages and environments reported in literature, and discusses the merits of simulation languages versus simulation libraries. The APSIS design requirements and decisions are discussed in Section 3.3, together with a description of the functional design of the application programming interface, the software architecture, and the hardware architecture. In Section 3.4, we describe the necessary extensions to the basic Time Warp method that are introduced to efficiently support simulations stemming from computational science. The specific implementation details of the key features of the Time Warp simulation kernel are presented in Section 3.5.

### 3.2 Parallel Discrete Event Simulation Environments

The migration process from sequential discrete event simulation to parallel discrete event simulation should be as smooth as possible. The user should concentrate his effort on the modeling process instead of being bothered with the details of parallel synchronization protocols. One of the most important design goals of parallel discrete event simulation environments is to provide a level of abstraction from these synchronization details in order to enhance the usability of PDES. The PDES environment hides the complexity of the synchronization protocol by providing pre-built simulation kernel(s) as well as development tools.

The various PDES environments can be described and compared by their constitutional components, namely modeling capability, programming framework, language features and library API, synchronization protocols, and system support and environment. The modeling capability determines how the physical system is modeled, i.e., the world view presented to the user (see Section 1.2.7). The programming framework (e.g., structured or object-oriented) incorporated by the PDES environment affects the development time and maintenance effort. For example, an object-oriented programming framework will reduce the development time of the simulation application. However, in general the object-oriented approach comes at a price of increased runtime overhead, and often results in slower execution speed as compared to structured languages such as C. Language features and library API provide a set of constructs to design simulation models. Runtime system support and simulation environment comprise different aspects such as logical process to processor mapping, performance evaluation, statistical information collection, and visualization and debugging capabilities.

An important feature of parallel simulation environments is whether the PDES facilities are incorporated in a (new) simulation language or in a runtime support library (Bagrodia 1998; Low et al. 1999). Simulation languages provide a full set of well-defined language constructs to design simulation models, whereas a library only provides a group of routines to be used with a
3.2 Parallel Discrete Event Simulation Environments

general-purpose programming language. Consequently, the conceptual modeling framework offered by the PDES environment is more prevalent in simulation languages than in libraries. Furthermore, a simulation language with the relatively high-level interface to the user allows for optimizations by a compiler that are cumbersome at the low-level interface of a library. PDES libraries, on the other hand, give the user more flexibility in controlling the simulation application in terms of the behavior of the underlying synchronization control. A knowledgeable user may fine-tune the options provided, but users must also note that if the options are not set correctly, the performance of the simulation may degrade significantly.

3.2.1 Languages

The influence of simulation-language research on the evolution of programming languages, indicates the importance of simulation to computing (Nance 1993). For example, the concept of object-oriented program design was first incorporated in the discrete event simulation language Simula 67 (Nygaard and Dahl 1978), the first object-oriented programming language. The object-oriented design methodology naturally accommodates the modeling activity in simulation, but appeared to be successful in a much broader modeling and design perspective to tackle the complexity of large software systems. The simulation languages discussed in this section are all object-oriented languages, with the exception of Parsec.

One of the most important benefits of simulation languages over simulation libraries is that simulation languages provide a more consistent framework or world view that typically makes it easier for the user to design a model. The simulation language constructs and semantics reflect the intended use and allow for a coherent transition from simulation model to simulation application. All languages discussed in this section support the process-oriented world view. A disadvantage of simulation languages is, however, that the user often needs to learn a new programming language, although there are simulation languages that are enhancements of familiar general-purpose languages. An additional advantage of enhanced general-purpose languages is their portability and a richer program development environment support.

The Yaddes system (Preiss 1989) provides an environment for constructing discrete event simulations. The principle features of the Yaddes system are the Yaddes simulation specification language and compiler, and the runtime libraries. The Yaddes language is a specification language in the style of Lex (Lesk and Schmidt 1979) and Yacc (Johnson 1979). The basic components of a Yaddes program are model specifications (describe general state machine), process specifications (create logical processes by instantiating models), and connections specifications (describe connections between logical processes). The Yaddes specification files are translated into C language programs that are then compiled and linked to the simulation runtime library. The runtime simulation libraries support the simulation execution mechanisms: sequential discrete event simulation, Chandy-Misra distributed discrete event simulation,
and Time Warp distributed discrete event simulation.

ModSim (Rich and Michelsen 1991) is an object-oriented simulation language based on Modula-2. It was developed under contract with the Army Model Improvement Program (AMIP) Management Office using the Jet Propulsion Laboratory's TWOS (Reiher 1990). A sequential version of ModSim, MODSIM II, was later developed and released by CACI Products. The ModSim simulation kernel was originally the TWOS operating system, while later development included also the SIM++ environment (see next section). Both simulation kernels for ModSim support exclusively optimistic simulation based on the Time Warp protocol. APOSTLE (Wonnacott and Bruce 1996) is a high-level object-oriented simulation language for PDES. APOSTLE runs on top of an existing optimistic simulator written in C++. The optimistic simulator currently used is based on the Breathing Time Buckets synchronization protocol (Steinman 1992). The APOSTLE language has support for granularity control that allows multiple events to coalesce so that the overhead of a single event is spread over many changes of state.

Bagrodia et al. (1998) developed the simulation language Parsec that provides an easy path for the migration of simulation models to operational software prototypes, implementation on both distributed- and shared-memory platforms, and support for visual and hierarchical model design. The simulation development environment supports a number of front ends for programming models: the C-based Parsec simulation language; a C++ library that can be interfaced with native C++ code; and the Parsec Visual Environment (Pave). A portable kernel executes Parsec programs on sequential and parallel architectures. Parsec programs may be executed in two modes—as (ordinary) parallel programs or as simulation models. The simulation kernel supports a sequential, three parallel conservative, and an optimistic synchronization algorithm.

The Fornax simulation language (van Halderen and Overeinder 1998; van Halderen et al. 1998) is a Java-based discrete event simulation language. The versatility of Java enables the expression of additional semantics to offer the same conceptual framework as simulation languages do. The object-oriented programming constructs in Java are extended to implement entity, event, and simulation time control objects. By extending the Java language a process-oriented simulation language is constructed, where method calls (also called event method calls) are now time-stamped interactions between entities. The interaction between entities by event methods requires that the method call (scheduling an event) and the actual method execution (handling an event) is decoupled. A number of simulation kernels are available: a sequential simulation kernel, a parallel simulation kernel exploiting parallelism on a multiprocessor using multiple lightweight processes, and a parallel simulation kernel with a distributed global clock. An optimistic simulation kernel is under development.

The performance of the sequential simulation kernel and the parallel simulation kernel using multiple lightweight processes in Fornax compares favorable with other Java-based simulation libraries and MODSIM III. A ring-
3.2 Parallel Discrete Event Simulation Environments

topology queueing network and an Ising spin system simulation were used for a quantitative performance evaluation. For the ring-topology queueing network simulation, the performance of Fornax is superior to MODSIM III up to 200 concurrently active queues (entities). The Fornax performance decreases for more than 200 concurrently active entities; this is due to increased multithreading overhead costs. For the Ising spin system simulation, Fornax outperformed the other Java-based simulation libraries. In the Ising spin system simulation there was no performance breakdown observed for more than 200 entities, as there are only a limited number of entities active at any instance of time.

Additional to the simulation kernel, the Fornax simulation environment provides a framework for visual modeling. For example, in computer architecture modeling and simulation, a set of predefined (functional) components, such as processor, memory, cache, bus, etc., can be made available in the visual modeling environment. The user designs the computer architecture using a graphical user interface and specifies the component parameters, e.g., memory size and access times. From this visual design a Fornax simulation is distilled and executed. Other promising features that are incorporated in the Fornax environment are the capabilities for Web-based simulation (Fishwick 1997) and agent-based simulation (Joshi et al. 1997).

3.2.2 Libraries

Libraries are typically composed of a simulation kernel implementing a (number of) synchronization protocol(s) and an application programming interface (API). The API provides an interface to the simulation kernel to create and initialize logical processes, to schedule events or interactions, and to finalize the parallel simulation. In this respect, simulation libraries fulfill a dualistic role. The library as such, allows a user to implement a parallel simulation in a general-purpose language such as C or C++, by calling the appropriate library functions via the API. The other role of libraries is its use to supply a simulation kernel for parallel simulation languages. Therefore, libraries and their functionality play a central role in parallel discrete event simulation.

Most PDES library environments reported in literature are research-oriented and based on optimistic protocols. In general, compared with optimistic protocols, conservative protocols can be implemented with less effort, and the optimization of the protocol is more application specific. For instance, lookahead information, which is crucial for a conservative protocol to be effective, is application dependent. Therefore, the most effective approach is for the programmer to use a general purpose parallel runtime system, and implement optimizations specific to the simulation application. The optimistic protocols are far more difficult to implement than conservative ones, due to the inherently complex and intractable nature of the Time Warp mechanism. Furthermore, the genericness of optimistic protocols makes that the method's effectiveness is less application independent, and can be optimized over a broad range of application behaviors. Thus, contrary to conservative protocols, the most effective
way is for the programmer to use a parallel simulation library that hides the implementation details of the Time Warp mechanism.

The list of PDES library environments discussed in this section is far from complete. In the presentation, the libraries are selected by their distinctiveness, and by the amount to which they integrate the various concepts and their use in PDES.

The Time Warp Operating System (TWOS) (Jefferson et al. 1987; Reicher 1990) is historically one of the first optimistic simulation environments based on the Time Warp protocol. TWOS is composed of two layers: the Time Warp layer and the kernel layer. The TWOS program interface hides the underlying kernel layer and hardware. Thus, the programmer cannot access raw hardware, nor can he use the underlying kernel layer beneath the level of the TWOS interface. Although all interactions with the virtual machine are performed through TWOS, it is not an interactive operating system. In its current form, it is linked with the simulation to form a single executable. Interesting features of the TWOS environment are dynamic object creation and dynamic load management. Dynamic object creation is a complicated problem. Objects are created dynamically upon requests of other objects in the simulation. Such a request may be part of an erroneous computation that will eventually be rolled back. Therefore, the dynamic creation may need to be undone. Either the actual creation must be delayed until the commit point, or the entire creation must be able to be undone. Dynamic load management allows the simulation to obtain good performance by dynamically balancing computational work over the nodes based on the effective utilization, which is the fraction of useful, committed work on the simulation.

SPECTRUM (Reynolds and Dickens 1989) is a test bed for designing and evaluating parallel simulation protocols. The test bed supports experimentation on a full range of protocols in a common environment by relying on filters exclusively to implement parallel protocols. By specifying filters for the actions like initialization, get-next-event, post-event, advance-time, and post-message, a specific protocol instantiation can be implemented. With respect to performance comparisons, it is recognized that the SPECTRUM test bed is no substitute for carefully crafted implementations of protocols for a given architecture.

The Georgia Tech Time Warp (GTW) (Das et al. 1994) system is specifically designed for cache-coherent shared-memory multiprocessors. The simulation kernel is based on the Time Warp mechanism, and is designed to support efficient execution of small granularity discrete event simulation applications. This design objective necessitates a simple program interface that can be efficiently implemented. More sophisticated mechanisms can be implemented as library routines. For example, GTW supports an event-oriented world view, but more complex world views such as process-oriented simulation can be built on top of the GTW kernel. A number of techniques are incorporated in GTW to enable efficient parallel execution of small-grained simulation programs. Some techniques are also applicable to message-based machines, but the most important, e.g., buffer management mechanism, GVT algorithm, and maintaining locality of state vector information, are specific to cache-coherent shared-memory
machines.

ParaSol (Mascarenhas et al. 1995) is a multi-threaded system for shared-memory and distributed-memory parallel simulation. ParaSol is designed to support active-transactions flow in process-interactive simulations. The idea here is to develop a process-style description of a transaction's activity as it flows through a system (similar to popular sequential simulation languages such as CSIM, GPSS, or SIMAN). Here, transactions are implemented via time-stamped threads which transparently migrate between processors to access model resources. The use of mobile threads has several advantages: locality of reference, potential for load balancing, and simplicity in application-level coding. On the other hand, the complexity of working with threads instead of messages is significant, and poses serious challenges to efficient kernel operation.

The primary objective of the Parsimony Project (Preiss and Wan 1999) is the development of a Java-based test bed for distributed, network-centric simulation. Parsimony exploits the features of Java to provide a flexible distributed simulation environment while hiding many of the simulation details from the user. However, different than Fornax, Parsimony requires explicit send and schedule method calls to model time-stamped interactions between entities. The Parsimony package includes three different distributed simulators. A distributed simulator is one in which a given user-defined simulation is executed using multiple Java Virtual Machines running on different host computers.

Other parallel simulation libraries not discussed here are SPEEDES (Steinman 1992), SIM++ (Baezner et al. 1994), SimKit (Gomes et al. 1995), WARPED (Radhakrishnan et al. 1996).

3.3 Design of the APSIS Environment

3.3.1 Requirements and Design Goals

The projected use of the APSIS simulation environment is primarily in the field of dynamic complex systems (Sloot et al. 1997) such as asynchronous cellular automata. Typical examples of asynchronous cellular automata are continuous-time Ising spin systems or population dynamics models. Without going further into the details of these application classes, we can characterize the application classes as data intensive, thus requiring efficient memory management, and in need of dynamic entity creation and deletion. These application requirements put constraints on the design of the APSIS environment.

The APSIS environment is a research vehicle for both optimistic simulation protocol design and evaluation, and parallel simulation development of dynamic complex systems. For effective parallel simulation development we need to hide the complexity of the optimistic synchronization protocol by providing a simulation language or library. The protocol design and evaluation asks for a flexible environment that is easily adapted to incorporate new concepts and is extendible to interface with the environment for evaluation of the
concepts. From a simulation modeling perspective, the model validation and simulation verification are important. This requires determinism: given the same input, a simulation should produce identical results, regardless of the number of processors or the mapping of the processes to processors used.

Operational requirements are transparent scalability and parallel efficiency. The envisaged parallel platforms range from massively parallel processors to networks and clusters of workstations. The simulations should be executable on varying number of processors and with different mappings of processes to processors without source code modifications. Over the various parallel platforms, efficient execution of the simulation should be guaranteed. Closely related to the transparent scalability and parallel efficiency, is the portability of the parallel simulation. Portability and efficient network support allows the development of the parallel simulation on a workstation, while ensuring that the simulation can be moved to a parallel processor for production runs.

3.3.2 Overview

The APSIS system contains a development and execution environment for parallel simulations. The parallel runtime executive is an optimistic simulation kernel based on the Time Warp protocol. Simulation of applications stemming from dynamic complex system requires efficient execution of small grain events. The basic world view supported by the simulation environment is event scheduling. More complex world views such as process-oriented (see Section 1.2.7) can be implemented on top of the simulation kernel. Furthermore, the data intensive characterization of dynamic complex systems motivates support for aggregation of entities in subdomains, allowing for data decomposition which closely relates to the modeling practices of the problem field. Although special considerations are taken for dynamic complex systems, APSIS is a general parallel discrete event simulation system that can be used for computer network simulations, personal communication services simulations, or aviation applications.

The simulation model adopted in APSIS is the well-known model where physical processes of a real-world system are represented by logical processes that interact with each other via time stamped event messages (see Section 2.2). A model of a physical process is the description of the state and the behavior of the associated logical process. The simulation system is than a network of instantiated models. The runtime support system of APSIS must therefore support the instantiation and the concurrent execution of models. For the instantiation of simulation models, the simulation environment provides initialization and virtual topology constructor functions. Problems with a regular interaction between the logical processes are easily instantiated with environment provided virtual topology constructors for, e.g., grid or torus topologies. Irregular topologies can be specified by the application user. The topology constructor also maps the logical processes to the parallel processors. For the concurrent execution, the simulation runtime support relies on message pass-
3.3 Design of the APSIS Environment

The message passing libraries abstract from the underlying hardware, MPP or cluster of workstations, and allows for portable program development that is scalable over the number of processing nodes. The supported programming model is single program, multiple data (SPMD) (Hwang 1993).

The PDES facilities of the APSIS environment are supported by a library. As APSIS is a research vehicle, the environment must be flexible and easily extensible. Languages provide a higher abstraction level, definitely preferable from a modeling perspective, but are less pliant to changes than libraries. Furthermore, knowledgeable users can fine tune the library parameters, such as state saving method, cancellation strategy, or optimism control, to attain the best performance with respect to the application class. The relative simple interface of the library can be efficiently implemented. If the various design dimensions have been crystallized, a production simulation kernel can be accommodated by a simulation language like Fornax (van Halderen and Overeinder 1998).

In Fig. 3.1 the APSIS architecture is shown. Four layers are depicted in the figure. The hardware layer is the parallel platform, which can range from a cluster of workstations to a massively parallel processor architecture. The communication libraries MPI, PVM, CK*, Panda†, etc., abstract from the specific hardware by providing a uniform, portable interface to the underlying parallel platform, i.e., the communication infrastructure. The APSIS library incorporates a number of interfaces to the various message passing layers. The simulation kernel schedules the events over the parallel architecture and coordinates the distributed logical processes using the message passing interface. The instantiation of the simulation model and the mapping of the logical processes by the topology functions makes use of the message passing layers. To the upper layers, the APSIS library supports an application programming interface for simulation initialization, event scheduling, and simulation finalization. The library is implemented in the C language, and supports both an API for C as well as C++.

3.3.3 The Application Programming Interface

The APSIS application programming interface includes a set of routines to design and implement parallel simulation applications. We organize the description of the interface routines according to their functional use: simulation instantiation, event scheduling and execution, virtual time management, memory management, and environment configuration and information.

*The Communication Kernel is a lightweight communication layer designed for simplicity and efficiency (Overeinder et al. 1995).
†In the APSIS environment, the PVM communication library (called PANPVM) implemented on top of Panda is used.
Figure 3.1: Overview of the APSIS parallel simulation environment architecture.

**Simulation Instantiation**

The APSIS initialization function instantiates the simulation model by creating the logical processes (LPs) making up the the parallel simulation. The virtual topology constructor functions allows the simulation programmer to describe the LP interaction pattern that is logically similar to the simulation model, thus abstracting from the particular parallel architecture, which can have different physical interconnection topologies. Tables 3.1 and 3.2 summarize the instantiation functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apsis_init</td>
<td>initialize the APSIS execution environment</td>
</tr>
<tr>
<td>apsis_finalize</td>
<td>terminates APSIS execution environment</td>
</tr>
<tr>
<td>apsis_abort</td>
<td>aborts APSIS execution environment</td>
</tr>
<tr>
<td>apsis_pid</td>
<td>process identifier of calling process</td>
</tr>
</tbody>
</table>

Table 3.1: Initialization, finalization, and environment routines.

The initialization function `apsis_init(int *argc, char ***argv)` initializes the APSIS execution environment and creates the logical processes (currently Unix processes, future development will include threads). The programming model is single program, multiple data (SPMD), hence all logical processes start the same executable. `apsis_init` accepts the `argc` and `argv` that are provided by the arguments to the ANSI C `main` function. The first argument in `argv` must be the number of logical processes in the parallel
3.3 Design of the APSIS Environment

simulation. This argument is used and discarded by \texttt{apsis\_init}. The other remaining arguments are available to the simulation program.

The function \texttt{apsis\_finalize(void)} cleans up all APSIS state and starts a termination detection algorithm to finalize the parallel simulation. The function \texttt{apsis\_abort(int \texttt{errcode})} aborts all the logical processes. The error code is returned to the Unix or POSIX environment.

The process identifier of a logical process can be determined with the function \texttt{apsis\_pid(void)}. The function returns an integer number \(i\), where \(0 \leq i \leq N - 1\) and \(N\) is the number of logical processes.

| \texttt{apsis\_cart\_create} | create Cartesian virtual topology |
| \texttt{apsis\_cart\_free} | free Cartesian virtual topology data structure |
| \texttt{apsis\_cart\_pid} | determines LP pid given Cartesian location |
| \texttt{apsis\_cart\_coords} | determines Cartesian coords given LP pid |
| \texttt{apsis\_graph\_create} | create graph virtual topology |
| \texttt{apsis\_graph\_free} | free graph virtual topology data structure |
| \texttt{apsis\_graph\_neighbors} | return the neighbors of an LP within a graph |

Table 3.2: Virtual topology routines.

The APSIS virtual topology functions are very similar to the MPI virtual topology functions. The generic virtual topology functions can be efficiently implemented on top of a message passing layer and allow the definition of a rich set of topologies.

Function \texttt{apsis\_cart\_create(int \texttt{ndims}, int \texttt{*dims}, int \texttt{*periods})} can be used to describe Cartesian structures of arbitrary dimension. The number of dimensions is specified in \texttt{ndims}, and the number of logical processes in each dimension in array \texttt{dims}. For each coordinate direction one specifies whether the process structure is periodic or not. For a 1D topology, it is linear if it is not periodic and a ring if it is periodic. For a 2D topology, it is a rectangle, cylinder, or torus as it goes from non-periodic to periodic in one dimension to fully periodic. The topology translation functions \texttt{apsis\_cart\_pid(int \texttt{*coords}, int \texttt{*pid})} and \texttt{apsis\_cart\_coords(int \texttt{pid}, int \texttt{ndims}, int \texttt{*coords})} provide a mean to determine the Cartesian coordinates of an LP and the pids of its neighbors.

\texttt{apsis\_graph\_create(int \texttt{nnodes}, int \texttt{*index}, int \texttt{*edges})} creates a graph topology. The three parameters \texttt{nnodes}, \texttt{index}, and \texttt{edges} define the graph structure. \texttt{nnodes} is the number of nodes of the graph. The nodes are numbered from 0 to \texttt{nnodes\_1}. The \(i\)th entry of array \texttt{index} stores the total number of neighbors of the first \(i\) graph nodes. The lists of neighbors of nodes 0, 1, \ldots, \texttt{nnodes\_1} are stored in consecutive locations in array \texttt{edges}. The array \texttt{edges} is a flattened representation of the edge lists. The total number of entries in \texttt{index} is \texttt{nnodes} and the total number of entries in \texttt{edges} is equal to the number of graph edges.

The definitions of the arguments \texttt{nnodes}, \texttt{index}, and \texttt{edges} are illustrated in the following example.
The input parameters are:

\[
\begin{align*}
nodes &= 4 \\
degree_0 &= (2, 3, 4, 6) \\
degree_i &= (1, 3, 0, 3, 0, 2)
\end{align*}
\]

Thus, \(degree_0\) is the degree of node zero, and \(degree_i - degree_{i-1}\) is the degree of node \(i\), \(i = 1, \ldots, nodes-1\); the list of neighbors of node zero are stored in \(edges[j]\), for \(0 \leq j \leq degree_0 - 1\) and the list of neighbors of node \(i\), \(i > 0\), is stored in \(edges[j]\), \(degree_{i-1} \leq j < degree_i - 1\).

The information function \(apsis_graph_neighbors(int pid, int max_neighbors, int *neighbors)\) returns the array \(neighbors\) with pids that are neighbors to the specified logical process.

### Event Scheduling and Execution

The basic simulation world-view incorporated by the APSIS environment is event-scheduling. In event-scheduling, a model's execution is viewed as a sequence of events, where an event can be represented by a message. The APSIS environment provides two functions to schedule and execute the events of the parallel simulation in causal order. A third (de-) scheduling function, \(apsis_cancel\), is discussed in Section 3.4.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(apsis_send)</td>
<td>schedule an event for future execution</td>
</tr>
<tr>
<td>(apsis_recv)</td>
<td>receive an event for execution</td>
</tr>
<tr>
<td>(apsis_cancel)</td>
<td>cancel an event</td>
</tr>
</tbody>
</table>

Table 3.3: Event scheduling and execution routines.

The event scheduling function is \(apsis_send(int dest, vtime ts, void *buf, int count)\), which sends a message \(buf\) of size \(count\) to destination \(dest\) with timestamp \(ts\). The timestamp of type \(vtime\) can be either an integer or a floating point value. The net effect of a call to this function is the scheduling of an event at LP \(dest\) for execution at simulation time \(ts\). The contents of the event message \(buf\) is non-specified by APSIS and depends on the simulation application.

A call to \(apsis_recv(void *buf, int count)\) retrieves the next pending event for execution. Indirectly, the retrieval of the pending event sets the
LVT of the LP to the timestamp of the event message. The event messages received over successive calls to apsis_recv are guaranteed in non-decreasing timestamp order, as the LPs must adhere to the local causality constraint.

The simplicity of the two routines apsis_send and apsis_recv is almost misleading, as it is the full complexity of the Time Warp protocol that realizes the local causality of the individual LPs. The details of the subtle interplay between the two routines are presented in Sections 3.3.4 and 3.5.1.

### Virtual Time Management

Table 3.4 shows the virtual time information inquiry functions. The function apsis_gvt(void) returns the current global virtual time known by the Time Warp kernel of the calling LP. apsis_lvt(void) returns the current local virtual time of the LP, that is the timestamp of the current event message being processed by the LP.

<table>
<thead>
<tr>
<th>apsis_gvt</th>
<th>global virtual time</th>
</tr>
</thead>
<tbody>
<tr>
<td>apsis_lvt</td>
<td>local virtual time</td>
</tr>
</tbody>
</table>

Table 3.4: Virtual time management routines.

### Memory Management

The APSIS environment does not include transparent, or implicit, state saving, thus the simulation application must explicitly save state changes for potential future rollbacks. The interface function to the APSIS state saving mechanism is shown in Table 3.5. The APSIS environment incorporates copy state saving and incremental state saving. (The design of incremental state saving in APSIS is further discussed in Section 3.4.2.)

| apsis_state_save | incremental state save |

Table 3.5: Memory management routines.

Independent of the selected state saving mechanism, copy or incremental state saving, the interface function is apsis_state_save(void *buf, int count). The function saves the (partial) state vector buf of size count bytes into a simulation kernel data structure.

### Environment Configuration and Information Routines

To allow for a flexible and extendible simulation kernel, a generic interface function to the Time Warp simulation kernel is provided by the apsis_attr_set(int attr, void *value) and apsis_attr_get(int attr, void *value) pair (see Table 3.6). These interface functions enable
knowledgeable users to fine tune the simulation kernel to their simulation application. The simulation kernel also has four debugging modes (in increasing detail of debugging messages), which can be set with \texttt{apsis\_debug(int mode)}. (The debugging mode can also be set via \texttt{apsis\_attr\_set}, but for convenience and historical reasons the \texttt{apsis\_debug} function is included.)

| \texttt{apsis\_attr\_get}  | retrieve library information |
| \texttt{apsis\_attr\_set}  | set library configuration   |
| \texttt{apsis\_debug}      | set library debug level     |

Table 3.6: Environment configuration and information routines.

Currently, the following simulation kernel attributes can be set: state saving method and virtual time window. The state saving method can be set to copy state saving or incremental state saving (default). The virtual time window controls the optimism of the Time Warp protocol by limiting the execution of events within a window of virtual time beyond the global virtual time (see also Section 2.4.5). Other simulation kernel attributes that can be requested are the number of processed events, committed events, and rolled back events.

### 3.3.4 The Software Architecture

#### The API – Time Warp Kernel Interaction

The application programming interface (API) described in the previous section, interacts with the Time Warp simulation kernel to orchestrate the distributed execution of events over the parallel platform. The instantiation functions are the interface to the Time Warp kernel module for logical process creation and topology definition. The event scheduling and execution functions, and the state saving function interact with the synchronization module of the simulation kernel. The functions to set kernel attributes work on specific parts of the Time Warp kernel, and the inquiry functions (including \texttt{apsis\_gvt} and \texttt{apsis\_lvt}) do not change the state of the simulation kernel.

#### The Time Warp Kernel

The APSIS Time Warp kernel is composed of three functional modules, namely the instantiation, synchronization, and GVT computation module (see Fig. 3.2). The modules work independent from each other; the instantiation module is only accessed during simulation startup and initialization. The modules do not interfere with each computation, but do asynchronously exchange information between each other. From Fig. 3.2 one can see that the instantiation and synchronization modules can be accessed via the API. The GVT computation module cannot be accessed by the application (with the exception of the \texttt{apsis\_gvt(void)} routine, which only returns the current GVT). All modules make use of the underlying message passing layer.
### 3.3 Design of the APSIS Environment

<table>
<thead>
<tr>
<th>API</th>
<th>instantiation</th>
<th>synchronization</th>
<th>GVT computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>PVM</td>
<td>CK</td>
<td>Panda</td>
</tr>
</tbody>
</table>

![Figure 3.2: Overview of the modules in the APSIS Time Warp kernel.](image)

**Instantiation**  The instantiation of the simulation model consists of APSIS environment initialization and logical process creation. The creation of logical processes relies on the facilities supported by the underlying message passing layer. Although the intrinsic of process creation are different for the various message passing layers, APSIS provides one single method for process creation that is translated to the particular message passing layer, see the next section on the message passing interface. With process creation, the message passing layer is also initialized. Apart from process creation, the instantiation initializes the data structures used by the synchronization and GVT computation modules.

The virtual topology construction, if requested by the simulation application, is translated to the underlying message passing layer. For message passing layers that incorporate the concept of virtual topologies, the virtual topology construction is effectively implemented using the message passing layer topology constructors. For message passing layers without the notion of virtual topologies, such as PVM, the virtual topology construction is implemented by translating the mapping of the logical processes to the processors according to the defined virtual topology.

**Synchronization**  The synchronization module implements the forward simulation and rollback protocol. The essential data structures for realizing the transparent rollback-based synchronization protocol for parallel simulation are the *input queue*, *output queue*, and *state queue*. The input queue, or event queue, data structure contains the events of the simulation in timestamp order. New scheduled events are inserted at the appropriate place, in timestamp order, in the queue. Closely associated with the input queue are the output and state queue, which contain respectively the event message sent and state changes due to the execution of an event (see for details of the data structure Section 3.5.1).

The interplay between scheduling and execution functions, and the local data structures of the Time Warp kernel is shown in Fig. 3.3. The receipt of an event (process `recv` in Fig. 3.3), retrieves the next pending event from the input queue. The execution of this event (process `exec`) can result in a number of state changes, which are stored in the state queue. The execution of the event can also induce a number (zero or more) of new events to be scheduled,
by sending event messages to their destination (process send). Copies of the event messages are stored in the output queue. All side effects resulting from the execution of an event are directly associated with that event. Thus, all state change and event message entries in their respective queues are linked with the “responsible” event in the input queue.

Figure 3.3: Scheduling and execution of events, and the interaction with the input, output, and state queue.

The rollback mechanism efficiently annihilates the scheduled events and undoes the state changes by using the associated data structures. Upon detection of a causality error, that is the receipt of a straggler event with a timestamp smaller than the current LVT, the rollback mechanism resets the LVT to the timestamp of the event directly before the straggler. All premature executed events with their associated side effects are nullified by running down the output and state queue, and sending out the anti-messages and replacing the state changes with their original values from the state queue. The rollback mechanism can be seen as the reverse loop of the normal forward simulation as shown in Fig. 3.3.

Global Virtual Time Computation  The Global Virtual Time (GVT) computation module coordinates a number of activities. First, of course, the GVT computation itself, that is the estimation of the smallest timestamp of the unprocessed events in the system. The second important task of GVT computation is fossil collection. And finally, GVT computation is used for distributed termination detection of the parallel simulation.

The GVT algorithm used in the APSIS environment is based on the algorithm proposed by Bauer et al. (1991, 1992). The underlying message passing layers provide error-free communication channels with first in, first out (FIFO) behavior. The APSIS message passing interface to the underlying message passing layer such as MPI, PVM, etc., enumerates all event messages through a certain communication channel, resulting in a unique event message number per destination. The GVT module locally administers the number of messages sent and received, and the minimum timestamp of an event message since the last GVT report. Periodically, the LPs send their local information and their LVT to the GVT manager, which deduces from the information the minimum of the transient message timestamps and LVTs (see Section 3.5.4). The GVT
3.4 Extensions to the Time Warp Kernel

The fossil collection task of the GVT module consists of freeing unused memory and committing irrevocable operations such as I/O. With each GVT update, the memory resources consumed by the input, output, and state queue are reclaimed. Because the LPs never rollback to a simulation time earlier than the GVT, it is safe to reclaim memory resources for events with timestamps less than GVT, together with their associated output and state queue entries. An important performance parameter of the GVT computation and fossil collection is the GVT update frequency. The three factors determining the optimal GVT update frequency are computational overhead, communication overhead, and memory usage. Computational overhead and communication overhead are minimal at low frequency, while optimal memory usage requires high frequency such that unused memory is reclaimed regularly. From experiments it appears that the communication overhead, that is the communication latency, is the most important factor in the determination of the optimal GVT update frequency. In case of low communication latency ($\pm 20\mu$sec), update frequency can be as high as 20 Hz, while in case of high communication latency ($\pm 1$ msec) the frequency can be as low as 0.5 Hz. Sensitivity analysis shows that frequency parameter setting is robust, that is, for large parameter ranges the GVT update performance is (sub-) optimal.

The GVT computation is also involved in termination detection of the parallel simulation. With the Time Warp protocol, the parallel simulation continues as long as there are events to be processed. If no events are pending for execution at an LP, the LVT of the LP will be set to $+\infty$. If all LPs will have their LVTs set to $+\infty$, and there are no transient event messages, then no more events are pending in the parallel simulation. With the next GVT computation, the new GVT value will be $+\infty$. This will trigger the termination phase of all LPs to finalize the parallel simulation.

The Message Passing Interface

The design of the APSIS message passing interface abstracts from the specific message passing layer used on the parallel platform. All Time Warp kernel routines access the message passing layer via the APSIS message passing interface. To support the APSIS library for a specific message passing layer, eight functions have to be supplied for program initialization and finalization, send and receive, and creation and freeing of Cartesian and graph topologies. The required send primitive is asynchronous (or buffered-mode in MPI terminology) and the receive primitive is nonblocking.

3.4 Extensions to the Time Warp Kernel

In this section we introduce two extensions to the Time Warp kernel that enlarge the effective application of the Time Warp protocol to highly dynamic
systems and simulations with arbitrary large states. First, we introduce a new primitive to retract scheduled future events. This allows us to actively remove simulation entities including their scheduled events. Second, the complication with applications with large states is solved by providing an alternative mechanism for copy state saving (checkpointing), called incremental state saving, which only saves the difference between two states.

### 3.4.1 Event Retraction

Many challenging simulations are not efficiently supported by the original formulation of the Time Warp protocol. For example, consider a population dynamics simulation model consisting of predators hunting for preys. Both predators and preys schedule their events for future activities, such as move, eat, and breed. Part of the modeled behavior of the predator is to catch a prey, resulting in the removal of the prey from the simulation. The elimination of the prey invalidates the previously scheduled events that initiate an activity of the non-existent prey. A convenient way of modeling this behavior is to have the predator “catch” event retract the original prey events in order to cleanup any trace of the prey. Although it is possible to simulate such activities without the ability to retract events, the availability of an event retraction primitive simplifies the simulation model, making it easier to understand and maintain (Overeinder and Sloot 1993).

At first glance, event message retraction seems to be quite similar to message annihilation in rollback, so message retraction could be implemented by sending the corresponding anti-message for the positive message that is retracted. However, there is one important difference: message retraction is part of the optimistic simulation and can potentially be rolled back, while message annihilation in rollback is part of the synchronization protocol which removes any reference to the event message in the parallel simulation. Thus, although the mechanism for message retraction is rudimentary available in the Time Warp protocol, we need to extend the annihilation mechanism to allow for rollback of a message retraction.

The new event retraction primitive is designed to interact with the existing annihilation mechanism without increasing the overhead of the Time Warp mechanism where the primitive is not used. Similar to rollback, the retraction primitive cancels an event message by sending the corresponding anti-message to the logical process that received the original message. But, in addition, a positive copy of the message is placed into a new data structure called the cancel queue that is associated with the logical process that invoked the retraction primitive. If the event that invoked the retraction primitive is rolled back, we send the positive copy of the message to the receiving process. Just as the output queue maintains the information necessary to roll back previous sent event messages, the cancel queue maintains the information to undo invocations of the retraction primitive.

Lomow et al. (1991) proposed a similar design for a mechanism for user-invoked retraction of events in Time Warp as described in this section. Agre
and Tinker (1991) introduced a retraction mechanism that interacts with the GVT computation and fossil collection mechanism. In their approach, event retraction is an irrevocable operation and is committed during fossil collection. Committing the event retraction consequently delays the execution until the GVT sweeps past the timestamp of the event retraction. An advantage of this approach is that a cancel queue is not necessary because the event retraction cannot be rolled back.

**The Retract Primitive**

The event retraction function \texttt{apsis\_cancel} is conceptually the inverse operation of the event scheduling function described in Section 3.3.3. The schedule primitive \( X = \texttt{apsis\_send}(\text{dest}, \text{ts}, \text{buf}, \text{count}) \) sends a positive copy of message \text{buf} with timestamp \text{ts} to logical process \text{dest}. The schedule function is extended to return an identifier \( X \), the descriptor of the message, which can be used to refer to the generated message, e.g., to retract it.

The function \texttt{apsis\_cancel}(\( X \)) retracts a previously sent message whose descriptor is \( X \). The retract function discards all effects of the event simulation. If the message is not yet executed, the event message is removed from the input queue. In case the event is executed, the destination logical process is rolled back, effectively undoing all side effects, the corresponding event message is removed, and the simulation continues as if the event was never scheduled. Because retraction is the inverse of event scheduling, \( \texttt{apsis\_cancel} (\texttt{apsis\_send}(\text{dest}, \text{ts}, \text{buf}, \text{count})) \) is equivalent to a no-op. The \texttt{apsis\_cancel} is only meaningful from a modeling perspective if the virtual time at which the retraction was issued is smaller than the virtual time at which the event is scheduled for execution. A sanity check tests whether this precondition is met before the retraction is executed.

**3.4.2 Incremental State Saving**

The design and implementation of incremental state saving (ISS) in the AP-SIS environment is an essential feature to effectively support the simulation of dynamic complex systems. The motivation and merits of ISS are extensively described in Section 2.4.4. In this section we present the details of the incorporation of ISS in the AP-SIS environment.

One of the design goals formulated in Section 3.3 specified efficient support for simulation applications stemming from dynamic complex systems, and in particular the class of asynchronous cellular automata (see for definition of asynchronous cellular automata Section 5.2). Asynchronous cellular automata are data intensive applications that easily consume all available physical memory. In terms of parallel performance, this application class falls in the category of memory-bounded speedup models (Sun and Ni 1993): the idea is to solve the largest possible problem limited by memory space. For data intensive simulation applications it is inefficient and even often infeasible to save copies of the complete state (even with periodic state saving). However, the execution of
an event in this application class only induces small changes local to the state vector, thus it is appropriate to save only the updated parts of the state using the incremental state saving technique.

The incremental state saving mechanism developed and implemented in the APSIS environment exploits the Markovian behavior of the state evolution. The parallel simulation of the dynamic complex system is spatially decomposed into a number of subdomains. Each subdomain is an aggregation of cellular automata represented by a logical process. The state of the logical process can be seen as a vector of the states of all the cellular automata it contains. The state of the logical process can now be written as \( s = (s(a_1), \ldots, s(a_N)) \), where \( N \) is the number of aggregated cellular automata in the subdomain.

The execution of an event \( e_i, 1 \leq i \leq N \), scheduled at automaton \( a_i \) for virtual time \( t \), only changes the state of automaton \( a_i \) by the locality of the transition rules of the cellular automata. Thus the execution of the event results in the new state vector \( s' = (s(a_1), \ldots, s'(a_i), \ldots, s(a_N)) \). Instead of saving the complete state vector, it is sufficient to save the old state \( s(a_i) \). The incremental state saving mechanism associates state \( s(a_i) \) now with event \( e_i \).

The APSIS environment incorporates copy state saving (CSS) and incremental state saving (ISS). The state saving method can be set with the apsis_attr_set simulation kernel interface function. With the apsis_state_save function a (partial) state can be saved to the state queue. This function is both used for CSS and ISS. The generic functions apsis_attr_set and apsis_state_save enable the simulation application to switch dynamically between CSS and ISS. However, it is the responsibility of the simulation application that with the specific state saving method the proper state information is saved with apsis_state_save. If the Time Warp kernel is in CSS mode, state recovery during rollback is accomplished by restoring the state vector directly from the state queue. In ISS mode, the simulation kernel reconstructs the state by restoring each saved state variable in reverse order, until the last event with a timestamp just before the event that caused the rollback.

Incremental state saving requires less state saving time and memory, at the cost of state reconstruction. The efficiency of the incremental state saving method compared to the copy state saving method depends on the rollback length and the percentage of the state that is modified due to the execution of an event. The tradeoff between copy and incremental state saving is discussed in detail in Section 2.4.4.

### 3.5 Implementation Aspects of the Time Warp Simulation Kernel

#### 3.5.1 Simulation Kernel and Data Structures

The central activity of the Time Warp simulation kernel is the management of the input queue, output queue, state queue, and cancel queue data structures. All synchronization, i.e., forward simulation and rollback, and fossil collection
operations work on these data structures. Figure 3.4 shows the overall design of the queue data structures in the simulation kernel.

![Diagram of queue data structures](image)

Figure 3.4: The APSIS Time Warp kernel input, output, state, and cancel queue data structure.

The input, output, state, and cancel queues are priority queues. A priority queue is a data structure for maintaining a set of elements, sorted according to an associated key value. Conceptually the queues are doubly linked lists, as shown in Fig. 3.4, although the implementation can be a calendar queue (Brown 1988), a heap (Cherkassky et al. 1996), or another data structure (Ronngren and Ayani 1997), as long as there is a previous–next relation between the queue elements. The doubly linked list data structure allows for almost effortless traversal of the queues in both directions, which is essential in forward simulation and rollback processing.

The input queue is the defining data structure, which maintains the event messages in timestamp order. The input queue is called *defining* with respect to the other queues because during insertion or deletion operations on the input queue, the elements in the input queue are maintained in timestamp order while the other queues are manipulated according to the changes to the input
queue. An input queue element has three references to its associated output messages in the output queue, its saved state changes in the state queue, and its event retractions in the cancel queue. In this data structure it is possible that a multiple number of output message, saved state changes, or event retractions are associated with one event execution. On the other hand, if the execution of an event does not result in any output message, state change, or event retraction, the respective reference directs to the element of the previous event. For example in Fig. 3.4, the second and third event message in the input queue have their cancel queue reference directing to the cancel queue element of the first event message in the input queue (see references labeled with (*) in Fig. 3.4).

The LVT points to the current event message being processed. All event messages received but not yet processed, have their associated output queue, state queue, and cancel queue references set to nil.

### 3.5.2 Synchronization

The simulation synchronization operations of the Time Warp kernel are forward simulation (event scheduling and execution), event retraction, and rollback. All operations manipulate the queue data structures in one way or another. Forward simulation and event retraction construct the data structures progressively, and rollback consistently reconstructs the data structures to an earlier virtual time.

#### Forward Simulation

Forward simulation consists of event scheduling and execution. The **apsis_recv** retrieves the next pending event for execution from the input queue. The operation sets the LVT (see also Fig. 3.4) to the current event timestamp. If the execution of the event results in one or more state changes, the original values of the state variables are copied to the state queue. The scheduling of a new event is accomplished with the **apsis_send** function, which sends an event message to the destination LP and stores a negative copy of the event message in the output queue. If the execution of the event is completed, the next pending event is selected with **apsis_recv**. The semantics and logical "correctness" of simultaneous events are defined and resolved by a tie-breaking mechanism imposed by the simulation kernel (Mehl 1992; Wieland 1997). Multiple events scheduled by an identical process for the same simulation time are executed in a FIFO fashion. For ties which occur as a result of scheduling from two different processes, priority is given to the event scheduled by the process with the smallest APSIS process identifier (**apsis_pid(0)**).

Optimism control (see Section 2.4.5) is also part of forward simulation. The optimism of the Time Warp protocol is controlled by limiting the execution of events within a window of virtual time beyond the global virtual time. The time window can be set with the **apsis_attr_set** simulation kernel interface. If the timestamp of the next pending event falls outside the virtual time window,
the execution of the event is throttled by blocking the apsis_recv function until the virtual time window is advanced (after a GVT update).

**Event Retraction**

Although event retraction can be considered to be part of the forward simulation, that is, it is part of the simulation application rather than the “invisible” simulation protocol, it is discussed separately for clarity.

For the implementation of the event retract function apsis_cancel, we can identify three situations. First, the LP which retracts the event is also the originator of the positive event message. Second, the LP which retracts the event is the destination of the positive event message. And third, the LP which retracts the event is neither the originator nor the destination of the positive event message.

In the first situation, with the message descriptor the original message can be found in the output queue (negative copy). In the second situation, the original message can be found in the input queue (positive message). In both situations, a negative copy of the event message is sent to the destination LP and a positive copy is stored in the cancel queue. The third situation is not implemented in our scheme, as there is no efficient method to retrieve a copy of the original message, and from a modeling perspective it is quite unlikely there is any need to support this situation.

We can give examples of these three event retraction situations in an individual-based population dynamics simulation (similar as discussed in Section 2.4.4). The population dynamics simulation, where predators and preys struggle for life, is spatially decomposed over a number of subdomains (assigned to different logical processes). The predators and preys have a certain (predefined) range of interaction, that is the distance a predator can see or smell a prey and hunt it down to eat it. The subdomains have a boundary region, which width is equal to the range of interaction distance. The first type of event retraction happens when a predator kills a prey within one subdomain: all future events scheduled for the prey have to be retracted. The second type of event retraction occurs when a predator in the boundary region kills a prey in the boundary region of another subdomain. Since changes in the boundary regions are communicated to neighboring subdomains, positive or negative copies of the event messages scheduled for the prey are available, and hence are retracted. The third situation, which is not implemented in the APSIS simulation kernel, would occur when a predator detects and kills a prey on a distance that is larger than the predefined range of interaction: this situation is precluded by the simulation model.

**Simulation Kernel Send/Receive Pair**

The APSIS interface functions for scheduling, receiving, and retracting events are handled by the send and receive functions in the simulation kernel. The simulation kernel send function routes the event message scheduled by the
The APSIS Time Warp Kernel

The APSIS send function call to the destination LP. The event message routing is optimized such that the delivery of messages at local destination LPs is actually a storage operation of the event into the input queue. Event messages with remote destinations are sent by the underlying message passing layer such as MPI or PVM. The simulation kernel receive function receives the event messages from the underlying message passing layer and stores the event into the input queue. The kernel receive function is implicitly called each time the apsis_recv function is called from the simulation application.

With the delivery of an event message, the event is stored into the input queue. The delivery of a local event message (or internal event) is directly handled by the send function, and the delivery of a remote event message (or external event) is processed by the receive function. The effect of input queue addition depends on the sign of the event message. A positive event message is inserted into the input queue at the appropriate place according to its timestamp. If a negative message (or anti-message) is inserted into the input queue, the negative message annihilates with the positive message in the input queue, thus removing the positive message. If the timestamp of the event message (either positive or negative) is smaller than the current LVT, a causality error has occurred and the rollback mechanism is triggered.

**Rollback**

Rollback is the basic synchronization mechanism to recover from causality errors. First the input queue is rolled back by resetting the LVT to the timestamp of the straggler event. Next, all side effects are undone by rolling back the output, state, and cancel queues. Rollback of the output queue is accomplished by sending the anti-messages to annihilate with the premature sent positive event messages. Similarly, rollback of the cancel queue incorporates sending the positive message, but an optimization is possible. If both the event retraction and the scheduling of the original event are rolled back, the positive message in the cancel queue and the anti-message in the output queue are removed and annihilated. Neither message needs to be sent to the receiving LP.

The rollback method of the state queue depends on the state saving mode: CSS or ISS (see Section 3.4.2). If the simulation kernel is in CSS mode, the rollback is accomplished by copying the saved state vector back in one single operation. If the simulation kernel is in ISS mode, the rollback of the state queue consists of running down the state queue and copying back each saved variable in reverse order.

**3.5.3 Fossil Collection and Irrevocable Events**

With every GVT update, the simulation kernel reclaims memory resources for events with timestamps less than GVT and commits irrevocable events for execution. With fossil collection, the memory resources for the events in the queue data structures are freed up to the last event with a timestamp smaller than the current GVT. The irrevocable operations are buffered during simulation.
and are committed and executed if the GVT sweeps past their simulation time. In Fig. 3.5 the circular I/O buffer is shown, with the two pointers GVT and LVT indicating the start and the end of the buffered data. During simulation all I/O is buffered into this circular buffer. If the simulation rolls back, the LVT pointer is moved back up to the buffered operation with timestamp smaller than the straggler timestamp, and hence the premature buffered operations are rolled back. If the GVT value is updated, the buffered events with timestamp smaller than the GVT are committed and executed, and the GVT pointer of the circular buffer is moved forward.

Figure 3.5: The circular I/O buffer for irrevocable operations. The two pointers GVT and LVT indicate the start and end of the buffered data. In (b) the LVT pointer swept past the circular buffer size.

3.5.4 The Global Virtual Time Computation

The APSIS global virtual time algorithm is based on the GVT algorithm proposed by Bauer and Sporrer (1992). The GVT algorithm is an asynchronous centralized algorithm where the GVT is computed by a central GVT manager that broadcasts a request to all LPs for the current LVT and while collecting those values perform a min-reduction. The GVT algorithm assumes error free communication channels with first-in, first-out behavior.

Let $S$ be the set of logical processes in the simulation, and $T_{ch}^{b\rightarrow a}(t)$ the minimum of timestamps of messages sent by $b$ but not yet received by $a$ (the transient messages over a channel). The GVT at any real time $t$ is defined as

$$GVT(t) = \min_{k \in S} \left( \min_{LVT^k(t)}, \min_{a,b \in S} T_{ch}^{b\rightarrow a}(t) \right).$$

In other words, GVT is the minimum of any LVT and any message that has already been sent but was not yet received up till now.

The definition of the extended local virtual time ELVT includes the minimum timestamp of transient messages for the logical process:

$$ELVT^a(t) = \min_{LVT^a(t), b \in S} T_{ch}^{b\rightarrow a}(t).$$

Using ELVT, we can now express the GVT as:

$$GVT(t) = \min_{k \in S} ELVT^k(t).$$
The APSIS Time Warp Kernel

To keep account of the transient messages, all messages through a certain channel are numbered, so we can use the following notation:

- \( n^{ba}(t) \): serial number of a message sent from \( b \) to \( a \) at real time \( t \), increasing with each message.
- \( r^{a-b}(t) \): number of messages received by \( a \) from \( b \) during time interval \([0, t]\).
- \( T^{ba}_m(n) \): timestamp of event message \( m \) with serial number \( n \).

The minimum timestamp of the messages sent by logical process \( b \) to logical process \( a \) during some time interval \([t_1, t_2]\) is given by

\[
T^{ba}_{\min}(t_1, t_2) = \min_{n^{ba}(t_1) < n \leq n^{ba}(t_2)} (T^{ba}_m(n)).
\]

We can now rewrite ELVT for real time \( t^a \) for logical process \( a \), as

\[
ELVT^a(t^a) = \min \left( \text{LVT}^a(t^a), \min_{b \in S} T^{ba}_{\min}(t^b, t^a) \right). \tag{3.1}
\]

Equation 3.1 states that we can determine \( ELVT^a \) at real time \( t^a \), if we know \( \text{LVT}^a(t^a) \) and the value of \( T^{ba}_{\min} \) during the interval \([t^b, t^a]\) (with \( n^{ba}(t^b) = r^{a-b}(t^a) \)) from each logical process \( b \in S \) that sends messages to \( a \).

The simultaneous reporting problem deals with information messages from different times \( t \) (the time for which to determine GVT). Given Eq. 3.1, we can determine a lower bound of GVT at real time \( t^b_2 \) despite the fact that we have not received any information from logical process \( a \) after \( t^b_2 \):

\[
\text{GVT}(t^b_2) = \min_{k \in [a,b]} (ELVT^k(t^b_2)) \geq \min \left( \text{LVT}^a(t^a_2), T^{ba}_{\min}(t^b_2, t^a_2), \text{LVT}^b(t^b_2), T^{ab}_{\min}(t^a_2, t^a_2) \right). \tag{3.2}
\]

In the APSIS GVT algorithm, each logical process \( k \) sends an information message to the GVT manager consisting of its \( \text{LVT}^k \), \( r^{k-a} \) for each incoming channel, and \( n^{k-b} \) and \( T^{kb}_{\min}(t^b_{k-1}, t^b_k) \) for each outgoing channel \( k \) is the current time and \( t^b_{k-1} \) the time of the previous information message). The GVT manager then applies Eq. 3.2, determining the \( t^a_1 \) ("message sent") times by comparing \( n^{a-b} \) and \( r^{b-a} \) for each channel \((n^{a-b}(t^a_1) \leq r^{b-a}(t^b_2)\) will give a valid \( t^a_1 \) time). The new GVT estimation will be distributed to the logical processes, which in turn can start their fossil collection and commit irrevocable operations.

### 3.6 Summary and Discussion

The APSIS simulation environment is designed and implemented to provide an experimental platform for design and evaluation of the Time Warp simulation protocol. The APSIS environment incorporates a number of extensions to efficiently support data intensive applications stemming from the research
field of dynamic complex systems. For ease in flexibility to introduce new protocol extensions and to realize an efficient interface to the parallel simulator, we have chosen to incorporate the parallel simulation functionality into a library instead of a simulation language. Moreover, interface functions to the library parameters enables knowledgeable users to select the appropriate state saving method, cancellation strategy, or optimism control attain the best performance with respect to the application class. APSIS is by design a portable simulation environment, which runs currently on a number of Unix platforms, such as Solaris, Linux, and BSD/OS, and with various communication libraries, like PVM, MPI, and the lightweight Communication Kernel (Overeinder et al. 1995). The APSIS application programming interface includes an interface to the C and C++ programming languages.

Event retraction and incremental state saving are introduced extensions to the Time Warp simulation protocol to accommodate the APSIS library for dynamic complex system applications. Event retraction enables the dynamic creation and deletion of simulation entities. A complication with dynamic deletion of simulation entities is the removal of scheduled events for those entities in the simulated system. The event retraction function is an API function that retracts, or cancels, a scheduled event similar to a rollback, but with the difference that the retract can also be rolled back, resulting in the reschedule of the original event. The incremental state saving facility provides a memory and time efficient state saving mechanism that is suited to data intensive applications which are spatially decomposed over the logical processors. The incremental state saving mechanism only saves the state changes due to the execution of an event, and upon rollback the saved variables are copied back in reverse order to reconstruct the original state vector.

Similar to other PDES libraries mentioned in Section 3.2.2, APSIS is research-oriented and based on an optimistic protocol. The event-oriented world view can be efficiently implemented such that minimal overhead is introduced. The APSIS library has efficient data structures to allow for fast and associative access to the input, output, state, and cancel queues. Together with other optimizations, such as local message delivery and efficient support to underlying message passing layers, an effective simulation environment for solving large scale problems is realized. Future enhancements can include adaptive optimism control, hybrid state saving, and simulation language support, for example by Fornax (van Halderen and Overeinder 1998).