dependent of the underlying synchronization mechanism while at the same time allowing maximum parallelism to be extracted from the model.

SimKit is a C++ class library that is designed for very fast discrete event simulation. SimKit presents a simple logical process view of simulation enabling both sequential and parallel execution without code changes to the application models. A brief overview of logical process modeling methodology innate within PDES is presented in section 2. Section 3 lists the issues that dictate parallel simulation language design considerations. The design philosophy adopted in the SimKit System is then outlined followed by the features of the SimKit System. Subsequent sections describe the SimKit simulation programming model, the SimKit language primitives and future developments envisioned.

2 LOGICAL PROCESS MODELING METHODOLOGY

In PDES, the physical system to be modeled is conceptualized as a system of interacting *physical processes (PPs).* A distributed simulation model of the physical system is represented by a topologically equivalent system of computational units called *logical processes (LPs).* Each LP is responsible for simulating events in a sub-space modeling the activities occurring in the corresponding PP. The interaction between PPs is modeled by the corresponding LPs communicating via timestamped messages. The timestamp represents the event occurrence time in the simulation. State transitions in the physical system are modeled by the occurrence of an event, i.e., the receipt of a timestamp message at the destination LP. Occurrence of an event may involve modifications to the state and/or the causing of new events in the future.

Distributed Simulation accelerates the execution of the simulation program by executing the computational units (LPs) concurrently on multiple processors. By nature, in a distributed simulation system, there is no notion of a global clock or a central event list. LPs execute in parallel by maintaining their own local clock and event list. A synchronization algorithm is used to ensure that the causality is maintained at each LP. Sharing of state between LPs is not possible. A separate synchronization mechanism should be provided to support consistent shared variables in PDES. Ghosh and Fujimoto (1991) uses the Space-Time memory system to support shared memory. Mehl and Hammes (1993) describe several mechanisms to support shared variables in distributed simulation.

3 SIMKIT DESIGN PHILOSOPHY

A plethora of parallel simulation languages have appeared in the last decade, each with differing design considerations. The languages include APOSTLE (Wonnacott and Bruce 1995) from DRA, U.K., Common Interface of OLPS (Abrams 1988, 1989), Maisie (Bagrodia and Liao 1990), ModSim (West and Mullarney 1988 and Rich and Michelsen 1991), MOOSE (Waldorf and Bagrodia 1994), SCE from MITRE (Gill, Maginnis, Rainier and Reagan 1989), Sim++ from Jade (Baezner, Lomow and Unger 1990, 1994), SIMA (Hassam 1991), RISE from RAND (Marti 1988) and Yaddes (Preiss 1989). The major differences between these languages are their approach towards

- the programming paradigm employed
- the underlying synchronization protocol
- the modeling world view
- run time configuration
- determinism, and
- efficiency

The desired characteristics of a parallel simulation language as outlined by Abrams and Lomow (1990) and Rajaei and Ayani (1992), include Simplicity, Modularity, Portability, Transparency, Evolvability, Efficiency, Scalability, Determinism, and Generality. **The primary goal of the SimKit System was to provide an event-oriented logical process modeling interface that facilitates the effortless building of application models for sequential and parallel simulation with high performance execution capabilities.**

The design philosophy of the SimKit Programmer's interface can be summarized as:

- *ease of use with the ability to reuse model components*
- *indifference towards the underlying simulation control system*
- *execution mode transparency*
- *encompassing a wide range of applications and user base*
- *efficient event-oriented logical process view of simulation*

The design features of the SimKit System include:

Object Oriented Programming Model: The SimKit library is implemented in the commonly-used general-purpose object-oriented programming language, C++. Object oriented techniques allow for reusable simulation models and software components to be developed. The iterative nature of a simulation life cycle is well supported by the object-oriented paradigm. Object-oriented techniques along with good software engineering principles can be used to design reliable simulation software, e.g., scoping via objects can be used to create complex simulations involving complex object class hierarchies.

By extending an existing language, the highly optimized library routines for parallel programming provided for a given architecture, can be directly used in the implementation of the underlying synchronization scheme. Moreover, the availability of debuggers, software engineering tools, etc. increase productivity and portability. Application programmers are not burdened with learning a new language and can concentrate on the system modeling task. The Simkit Class library includes just three classes: `sk_simulation` for simulation control, `sk_lp` for modeling application sub-space behavior and state transitions, and `sk_event` to model the interactions between the logical processes.

Transparency: The design of the Parallel SimKit system highlights the need for hiding issues pertaining to the underlying synchronization algorithm, *wherever possible*. Time Warp relies on a rollback mechanism, hence great care was taken in providing primitives for error handling, output handling and dynamic memory management. However, the application programmer is responsible to efficiently use the various state saving mechanisms provided. State Saving is an artifact of the need to support the rollback mechanism in Time Warp and for the present is left under the jurisdiction of the application programmer as a trade off for efficiency. This situation is incompatible with the other goals of the system and we are vigorously exploring ways of providing transparent (and efficient) state saving. Space precludes a full discussion of the automated state saving schemes explored. State Saving in the sequential Simkit system is completely ignored

State restoration during rollback is transparent to the application program. SimKit libraries include a comprehensive set of pseudo-random number distributions with efficient state saving of the seeds. LP allocation to processors in the parallel system is static and may be optionaly specified by the programmer.

Evolvability: Parallel SimKit has been implemented atop a Time Warp executive interface called the *Warpkit Interface*. Great care was taken to implement a minimal clearly-defined interface to the WarpKit Kernel. The interface comprises just three classes, namely, `wk_simulation`, `wk_lp` and `wk_event`. The SimKit classes in the parallel implementation) are derived from and mirror WarpKit classes. The Time Warp related quasi- operating system services like *event-delivery, rollback, and commit* are provided as virtual functions in the `wk_lp` class. This minimizes the impact of modifications to the underlying synchronization system on the Programmer's Interface and consequently the application itself. Evolving schemes and feature extensions can be easily integrated into the simulation control engine without affecting the SimKit programmer's interface.

Efficiency: SimKit is a high performance simulation language for event-oriented logical process view of simulation. SimKit supports the efficient sequential and parallel execution without code changes to the application model. The sequential simulation control system is a highly optimized simulator that uses the splay tree implementation of the future event list. The parallel Time Warp system is capable of exploiting the inherent parallelism in applications with very low event computation granularites, in the order of a few microseconds on current RISC processors.

The parallel run time system consists of a variation of the Time Warp protocol optimized for execution on shared memory multiprocessors. The global control algorithms are asynchronous in nature, with minimal locking of shared structures. The design of global control algorithms demonstrate a high degree of scalability, i.e., minimal performance degradation with an increase in the number of processors used. The problem of state saving and restoration in Time Warp is addressed by providing a grab bag of state handler objects that are fine tuned for saving integers, floats, doubles, pointers, variable length state and block states incrementally.

The event-oriented view of simulation enables efficient scheduling of events via invocation of the corresponding LP's event-processing member function rather than the costly context-switching approach required in a process-oriented approach. Also, the memory requirements are lower as LP's are active only for the duration of an event. Event-oriented views also allows for the dynamic creation and destruction of LPs.

Generality: SimKit provides a general purpose simulation interface. Both a sequential and a parallel execution mode are supported. The sequential executive performs well on a variety of platforms and facilitates debugging and testing. The per event scheduling overheads for the sequential executive on an SGI

Power Challenge, IRIX64 Release 6.0 IP21 mips multiprocessor architecture using the AT&T C++ compiler version 3.2.1 with the `-O2` compiler option, is 2.2 microseconds. The per event overheads for the parallel executive is 8.6 microseconds.

4 SIMKIT PROGRAMMING MODEL

The SimKit Programmer's Interface consists of three classes, one type and several convenience functions and macros. The three classes are: `sk_simulation`, `sk_lp`, and `sk_event`. The one type is `sk_time`. Also a library for random number generation and distributions is provided which includes classes for uniform, normal, exponential, geometric, binomial, Erlang and Poisson distributions.

Due to the scoping rules, base class names are visible to the application program and may introduce errors if the programmer uses these set of identifiers. Application programmers are warned not to use identifier names with a prefix `sk_`.

The simulation model is constructed by deriving LPs from the `sk_lp` class and messages (or events) from the `sk_event` class. The `main` function is in the modelers domain and may be used to incorporate third party tools like a lex/yacc program code to parse the simulation input parameters. The user instantiates **one** instance of the `sk_simulation` class. This object provides the initialization interface to the run time simulation kernel. The type `sk_time` is used to represent simulation time and supports standard arithmetic operators associated with the type *double*. The convenience functions and macros are provided for easy access to simulation structures like *current LP, current event, and current time.*

The modeler specifies an LP's activity via the `sk_lp::process` pure virtual member function. The kernel delivers an event to a LP by invoking this function with the event object as a parameter. Typically, the process function is coded as a large case statement with the event type specifying the branching of control. Two other virtual members provided are the `sk_lp::initialize` and the `sk_lp::terminate`. These may be optionally defined to initialize the LP and for doing wrap up LP work (e.g., printing LP statistics and reports).

The derived message types may contain other application specific data (besides event type) that are communicated between the LP's. A message is created using the SimKit overloaded event's *new* operator. The `sk_event::send_and_delete` is the only event synchronization primitive that is necessary in the event-oriented logical process view. The WarpKit kernel is implemented on a shared memory architecture (Fujimoto 1990b). The LP's event lists reside in shared memory. Message based communication, i.e., sending an event, is accomplished by allowing logical processes to directly access and modify another logical processes' event list. Mutual exclusion to event list is provided by software locks.

4.1 Time Warp Constraints

The following constraints extraneous to logical process modeling view are native to the parallel synchronization mechanism. Output handling and error handling should be rollback-able. Hence the modeler should use the member functions provided in the `sk_lp` class. General purpose memory management is another facility that is provided via the `sk_lp` class. The request for dynamic memory does not return if the allocation request cannot be satisfied due to system resource exhaustion. To support parallel simulation in particular, the simulation execution phase has been divided into 6 cleary defined phases.

State Saving calls must be explicitly programmed within the LP's process function. The grab bag of state savers described by Cleary, Gomes, Unger, Xiao and Thudt (1994) are provided within the `sk_lp` class for incrementally saving basic data types, variable length data sizes and block state. Options of using Copy State Saving instead of Incremental State Saving mechanisms allows simulations programs to be debugged for correctness before optimizing using ISS mechanisms.

Constraints of the Time Warp mechanism and the parallel executive in particular include :

- event ownership is retained by the kernel when a LP processes the event (saved in the input queue until fossil collection)
- lack of any input facility during simulation execution
- lack of interactivity during simulation execution, output occurs (usually in bursts) when Time Warp commits events
- prohibition against using global memory during simulation execution
- prohibition against LPs invoking member functions of other LPs directly, during simulation execution

4.2 Run Time Configuration

Facilities for collecting statistics about the simulation execution, for tracing simulation execution and debugging are provided. Two control mechanisms for tracing and debugging are provided. One is via compilation flags that conditionally compile into tracing and debuging code and the other is via run time flags that specify which bits of the trace and debug code to activate. Run time flags may be set via configuration file, the command line or the `sk_simulation` interface.

5 EXECUTION PHASES

Executions starts and ends with a single thread of control executing on a single processor. The six phases are listed below:

1. Program Initialization: The function `main` begins execution. Application model command line arguments are stripped off and the `sk_simulation` object is instantiated.

2. SimKit and Model Global Initialization: In this phase the `sk_simulation` object is initialized. Command line arguments are stripped off and SimKit configuration parameters read from an external file (filename may be specified via command line argument). All model LPs are instantiated and allocated to processors. Allocation of LPs to processor is static and may be optionally specified by the modeler via the LPs constructor. Global application data structures (READ only variables) are built during this phase. This phase ends with passing control to the simulation run time system.

3. Logical Process Initialization: During this phase, each LP's initialize member function is invoked once. Simulation time does not advance in this phase and no events are received. This phase is used to send out seed events to start the simulation.

4. Simulation Execution: Execution is under Time Warp control. Events are delivered to LPs by invoking the `sk_lp::process` function. Here the constraints mentioned in the earlier section are enforced. This phase terminates either normally due to end of simulation or abnormally due to an error in the simulation (committed).

5. Logical Process Termination: Each LPs `terminate` function is invoked. This phase is typically used for reporting LP specific statistics.

6. Simulation Clean-Up: The simulation run time system returns control back to the `main` function. This phase is generally used to tally statistics and output final reports.

6 FUTURE DEVELOPMENTS

Future enhancements within the SimKit system will be devoted to:

- providing determinism in the parallel system
- enabling dynamic creation and destruction of LPs
- automating the state saving process in simulation modeling
- providing libraries of abstract data types such as queues, linked lists, etc
- developing tool kits customized for simulation of specific applications
- providing a development environment that enables visualization of the simulation including forward and backward execution.
- enhancing the run time system with automatic load balancing abilities

7 SAMPLE PROGRAM FRAMEWORK

This is a sample application model comprising of Players arranged in a ring topology. Each Player juggles a green ball while tossing a red ball to his/her neighbor, per unit time. The model counts the number of balls seen by a Player. Each player is modeled by a LP and the arrival of a model at a LP is modeled by a event. The activities modeled in a LP are the update of a counter (state variable) and the passing of the ball to itself/neighbor by sending an event.

```
// Ball class derived from sk_event
class Ball: public sk_event {
 public:
   Ball( int color )
     : sk_event(), color_(color) { /* NULL */ }
   const int ball_color() const { return color_; }
 private:
   int color_;
};

// Player derived from sk_lp
class Player : public sk_lp {
 public:
   Player()
     : sk_lp() green_cnt_(0), red_cnt_(0)
       { /* NULL */ }

   void set_dest( const Player * dest )
     { dest_ = (Player *)dest; }

  void initialize();
  void process(const sk_event *);
  void terminate();

 private:
  enum { GREEN = 0, RED };
  int green_cnt_, red_cnt_; //LP's state
  Player* dest_; // LP's constant state
};

void Player::initialize()
{ Ball* ball;
  sk_time snd_ts, rcv_ts;

  rcv_ts = (sk_time)1.0;
     // send the startup messages
  ball = new Ball( GREEN );
  ball->send_and_delete(this,rcv_ts);

  ball = new Ball( RED );
  ball->send_and_delete(dest_,rcv_ts);
}

void Player::process(const sk_event *ev)
{
  Ball* ball;
  sk_time Clock = ev->recv_time();

  switch ( ((Ball *)ev)->ball_color() ){
    case GREEN:
      SaveInt( & green_cnt_ );
      green_cnt_ ++;
      ball = new Ball( GREEN );
      ball->send_and_delete(this,Clock+1.0);
      break;
    case RED:
      SaveInt( & red_cnt_ );
      red_cnt_ ++;
      ball = new Ball( RED );
      ball->send_and_delete(dest_,Clock+1.0);
      break;
  }
}

void Player::terminate()
{
  printf("Player[%d]: Grn[%d] Red[%d]\n",
    lp_num(), green_cnt_, red_cnt_ );
}

int main(int argc, char **argv)
{  Player *first, *next, *last;
   sk_time total_time;

   int num_players, i,
     // Phase I  Kernel Instantiation
   sk_simulation sim_kern;
   num_players = atoi(argv[1]);
   total_time = atof(argv[2]);
   argc = argc - 2;


     // Phase II
   sim_kern.initialize(argc, argv[3]);
   sim_kern.max_in_event_size(sizeof(Ball));
   sim_kern.set_end_time( total_time );

     // LP instantiation
   last = first = new Player();
   for (i=1; i<num_players; i++) {
       next = new Player();
       last->set_dest( next );
       last = next;
   }
   last->set_dest( first );

     // Phase III
   sim_kern.start_simulation();

     // Phase IV & V  Kernel control

     // Phase VI control returns
   ::printf("Simulation ends\n");
}
```

8 CONCLUSION

The SimKit system was designed to present a simple and elegant logical process view of discrete event simulation enabling both sequential and parallel execution without code changes to application models. SimKit was developed in C++ and exposes the benefits of object-oriented simulation, namely modularity, improved reliability and reusability. Object Oriented Simulation is also in accord with the logical process modeling methodology.

In this paper, a brief overview of the logical process modeling methodology is presented. The C++ class libraries that constitute the SimKit System were outlined. The simulation environment along with the underlying sequential and parallel simulation executives were detailed. The paper closed with a tutorial on how to build object oriented simulation models in SimKit.

SimKit is currently being used by researchers, simulationists and instructors for parallel simulation experiments. Continuing development of SimKit, *http://bungee.cpsc.ucalgary.ca/*, is devoted to building specialized class libraries for modeling communication network, computer system, transportation system and general purpose libraries like streams, linked lists, trees, etc.

ACKNOWLEDGMENTS

We would like to thank the industrial sponsors of the TeleSim Project including AGT Ltd, Jade Simulations, Newbridge, SAIC, Siemens and Stentor. We also acknowledge the continued support of the National Science and Engineering Research Council of Canada.

REFERENCES

Abrams, M. 1988. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, Eds. M. Abrams, P. Haigh, and J. Comfort, San Diego, California, 210–219.

Abrams, M. 1989. A common programming structure for Bryant-Chandy-Misra, Time Warp, and sequential simulators. In *Proceedings of the 1989 Winter Simulation Conference*, Eds. E. MacNair, K. Musselman, and P. Heidelberger, Washington D.C., 661–666.

Abrams, M., and G. Lomow. 1990. Issues in languages for parallel simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, Ed. D. Nicol, San Diego, California, 22(2):227–228.

Baezner, D., G. Lomow, and B. Unger. 1990. Sim++: The transition to distributed simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, Ed. D. Nicol, San Diego, California, 22(2):211–218.

Baezner, D., G. Lomow, and B. Unger. 1994. A parallel simulation environment based on Time Warp. In *International Journal in Computer Simulation*, 4(2):183–207.

Bagrodia, R., and W. Liao. 1990. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, Ed. D. Nicol, San Diego, California, 22(2):205–210.

Cleary, J., F. Gomes, B. Unger, Z. Xiao, and R. Thudt. 1994. Cost of state saving and rollback. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, Eds. D. Arvind, R. Bagrodia, and J. Lin, Edinburgh, Scotland, U.K., 24(1):94–101.

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

Chandy, K., and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206.

Ferscha, A., and S. Tripathi. 1994. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, Dept. of Computer Science, University of Maryland, College Park, MD 20742.

Fujimoto, R. 1990a. Parallel discrete event simulation. *Communications of the ACM*, 33(10):33–53.

Fujimoto, R. 1990b. Time Warp on a shared memory multiprocessor. *Transactions of the Society of Computer Simulation*, 6(3):211–239.

Fujimoto, R. 1993. Parallel and distributed discrete event simulation: Algorithms and applications. In *Proceedings of the 1993 Winter Simulation Conference*, Eds. G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, Los Angeles, California, 106–114.

Fujimoto, R., and D. Nicol. 1992. State of the art in parallel simulation. In *Proceedings of the 1992 Winter Simulation Conference*, Eds. J. Swain, D. Goldsman, R. Crain, and J. Wilson, Arlington, Virginia, 25:246–254.

Ghosh, K., and R. Fujimoto. 1991. Parallel discrete event simulation using Space-Time memory. *Proceedings of the International Conference on Parallel Processing*, 3:201–208.

Gill, D., F. Maginnis, S. Rainier, and T. Reagan. 1989. An interface for programming parallel simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Eds. B. Unger, and R. Fujimoto, Tampa, Florida, 21(2):151–154.

Hassam, R. 1991. SIMA: A parallel simulation environment. Technical Report TRITA-TCS-91007, Dept. of Telecomm. and Computer Science, The Royal Institute of Technology, S–10044, Stockholm, Sweden,

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

Marti, J. 1988. RISE: The RAND Integrated Simulation Environment. In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, Eds. B. Unger, and D. Jefferson, San Diego, California, 19(3):68–72.

Mehl, H., and S. Hammes. 1993. Shared variables in distributed simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93)*, Eds. R. Bagrodia, and R. Jefferson, San Diego, California, 23(1):68–75.

Page, E., and R. Nance. 1994. Parallel discrete event simulation: A modeling methodological percspective. In *Proceedings of the 8th Workshop on Paral-*