Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

AN OBJECT-ORIENTED ENVIRONMENT FOR FAST SIMULATION USING COMPILER TECHNIQUES

Yiqing Huang
Ravishankar K. Iyer

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main, Urbana, IL 61801, U.S.A.

ABSTRACT

In this paper, an efficient simulation environment that utilizes compiler techniques to speed up simulation is presented. The method is based on the utilization of flexible, process-oriented modeling and the event-oriented simulation, which provides minimum run-time system overhead. A compiler is implemented to transform a process-oriented model to an event-oriented model to completely eliminate the context-switches that are normally inherent in process-oriented simulation tools. Three different systems are simulated using the proposed method and are compared with DEPEND, a process-oriented dependability simulation tool. Results show that the simulation time is reduced significantly. The techniques proposed are general and can apply to process-oriented simulation models to speed up the simulation.

1 INTRODUCTION

Process-oriented simulation has long been employed for computer system simulation (Franta 1977, Law and Kelton 1991). It offers modeling flexibility over event-oriented simulation. However, context-switching often exists in its run-time system, and the overhead may lead to serious performance degradation compared with event-oriented simulation. This paper proposes compiler-assisted transformation techniques to eliminate context-switching overhead by transforming a process-oriented model into an event-oriented model. A simulation environment based on the techniques is implemented to evaluate the performance of the method. The major contributions of the paper are summarized in the following:

- Language constructs are designed to build process-oriented models and to facilitate transformation to event-oriented models.
- Transformation of process-oriented models into event-oriented models is automatic based on compiler techniques.
- Run-time support is implemented based on event-oriented simulation that completely avoids context-switching overhead.
- The technique proposed is general, and the performance improvement over machine-dependent simulation package is significant.

The remainder of the paper is organized as follows. Section 2 presents the motivation behind the proposed method and describes related work. Section 3 discusses the process-oriented modeling interface. Section 4 describes in detail the compiler-assisted transformation. Performance results and analysis are given in Section 5. Conclusions and future directions for this work are addressed in Section 6.

2 MOTIVATION

In process-oriented simulation, the components of a system to be simulated are identified and modeled as a collection of simulation processes. Each simulation process describes the behavior of a component such as communication, resource management, or error propagation. Coroutines (Jain 1991) or light-weight threads (Cooper 1988, Schwetman 1986) are usually incorporated in the run-time system to implement simulation processes. Context-switching is used to switch between coroutines or light-weight threads. Process-oriented simulation packages based on coroutines or light-weight threads in the literature include CSIM (Schwetman 1986), YACSIM (Jump 1993), and DEPEND (Kumar 1993).

Event-oriented simulation (MacDougall 1987) models a system with explicit events identified from the dynamic

behavior of the system. Those events are implicitly expressed in the process-oriented simulation using simulation constructs that model server queues, message exchange, or semaphores. Event execution in the process-oriented simulation requires activating a simulation process. Reactivation and suspension of simulation processes are implemented using context-switching. Events are explicit in the event-oriented simulation, therefore, context-switching is not necessary. However, more efforts are required to develop an event-oriented model; therefore, event-oriented simulation is only appropriate for small to medium sized problems. SMPL (MacDougall 1987) is an early event-oriented simulation package. SIMEX, developed at University of Minnesota, is another event-oriented package. It allows functions to execute like threads without context-switching. However, it is necessary for the user to manually break the modeling code into separate pieces to execute like threads.

Utilizing the flexibility of process-oriented modeling and the efficiency of event-oriented simulation motivates the development of a simulation environment that possesses both characteristics. Each process is transformed to a set of event functions performing the same task as the process. Thus, a mechanism to properly generate the event functions is essential. In this paper, compiler techniques are adopted to perform the transformation.

3 PROCESS-ORIENTED MODELING

3.1 Hierarchical Process-oriented Modeling

To help users to construct process-oriented models, we provide an object-oriented hierarchical modeling framework, shown in Figure 1.

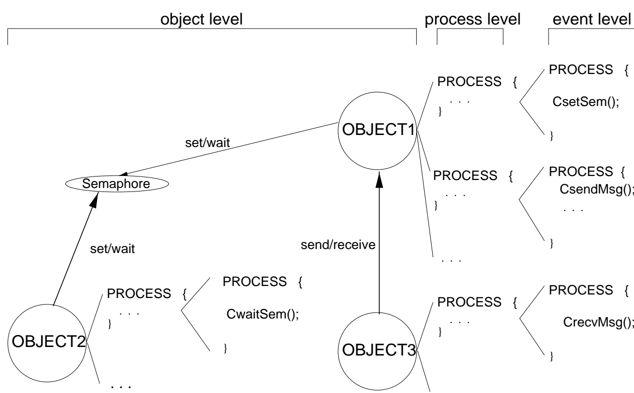

Figure 1: Process-oriented Interface Hierarchy

A process-oriented model is built using three levels: object level, process level, and event level. The model consists of a set of objects at the highest level. Each object contains a collection of processes. Each process employs simulation constructs to describe events implicitly. Each object can represent one physical component in the simulated system, e.g., a memory module in a computer system. To describe various behavior associated with each object, each object contains a set of processes. A process here is the same as a process in a process-oriented simulation language: represents activities occurring in the simulated system. For example, in the M/M/1 queue, the arrival of jobs to the server can be modeled as a process, which describes the behavior of job arrivals. If a triple-modular redundant system is simulated, the voter behavior can be modeled as a process. Each process employs simulation constructs to implicitly specify events occurring in the process. The simulation constructs constitute the event level of the modeling hierarchy. They model delay events, message exchange events, synchronization events, and server-related events. The object level, process level, and event level constitute the modeling hierarchy. The user employs this hierarchy to build simulation models in the process-oriented view.

In Figure 1, a user model is constructed utilizing this hierarchical framework. The three levels are shown in the figure. The model consists of three objects: **OBJECT1**, **OBJECT2**, and **OBJECT3**. **OBJECT1** and **OBJECT2** synchronize through their processes using the semaphore related simulation construct set/wait. **OBJECT1** and **OBJECT3** communicate through their processes using the message exchange simulation construct send/receive.

3.2 Modeling Hierarchy Implementation

The modeling hierarchy is implemented using C++. C++ objects are used to implement the object level. To specify processes in an object, **PROCESS** member functions are designed to extend C++ object to describe process behavior. The extended C++ objects, the **PROCESS** member function, and the simulation constructs are described in Table 1. The names of simulation constructs start with C. **Cdelay** can delay execution of a process for some simulated time or delay process execution until certain conditions are satisfied. **CsendMsg** and **CrecvMsg** are used to simulate process communication. **CsetSem** and **CwaitSem** synchronize processes. **Cserver**, **CreqServer**, and **CendServer** simulate the behavior of computer servers. Each simulation construct represents certain event.

4 MODEL TRANSFORMATION

Model transformation is conducted on the process-oriented models described in the previous section. Figure 2 shows a high-level picture of the transformation. The left part of Figure 2 is the process-oriented model. Each object in the process-oriented model is transformed into an object

Table 1: Extended C++ Objects, **PROCESS** Member Function, and Simulation Constructs

Name	Description
Extended C++ object	Models the simulated system. They usually correspond to active components of a physical system.
PROCESS member function	Models simulation process behavior.
Cschedule	Schedules object invocation based on specific time or occurrence of certain events.
Cdelay	Models the simulation time elapse or rescheduling of a simulation process when certain condition are satisfied.
CsendMsg	Simulates message sending. It specifies the destination process which receives the sent message.
CrecvMsg	Simulates message receiving. It specifies the source process of the message. The receiving process is suspended until the message arrives.
CwaitSem	Simulates synchronization. Simulation processes can wait on semaphore.
CsetSem	Opposite operation of CwaitSem
Cserver	Simulates the request arrival / service activities of a server.
CreqServer	Simulates the request of service of Cserver
CendServer	Simulates the end of service of Cserver

in the event-oriented model. Transformation for processes and simulation constructs of a process-oriented model is implemented using containers and event functions. A container is a high-level object designed to contain the transformed code of a process, as shown in Figure 2. For each process, behavior is modeled using simulation constructs that express events implicitly. The event-oriented model requires these events to be explicit so that scheduling can be performed by the event-oriented simulation engine. Transformation identifies events inside processes based on simulation constructs. For each simulation construct in the process-oriented model, there are construct associated activities. After the transformation, events are obtained and simulation construct associated activities are transformed into event associated activities. We design event functions to contain the generated events and event associated activities. For each event, an event function is generated to contain the event and the event associated activities. Each event function also contains scheduling information for the event functions in the container. Each container contains a set of event functions, and events associated with a process are kept in the same container. Based on the transformation, each container accomplishes the same task as the original process.

4.1 Compiler

To automate the transformation, a compiler is developed to perform the following four major functions: 1) Generate an abstract syntax tree, 2) Interpret simulation constructs, 3) Construct an event function flow graph based on simulation constructs and the control flow of the process model, and 4) Generate code for the event model.

Syntax tree construction and code generation are standard compiler techniques and will not be discussed here. To conduct transformation, the compiler first recognizes simulation constructs and analyzes the control flow in the **PROCESS** body. It then generates event functions and containers of those functions, as shown in Figure 2. The object boundary are unchanged from the original process model. The communication between the two processes is preserved using the event functions in the container.

4.2 Interpretation for Simulation Constructs

Simulation constructs are categorized into two classes: TYPEI and TYPEII. TYPEI includes **Cschedule, CsendMsg, CsetSem, Cserver**, and **CendServer**. These simulation constructs are translated directly to library calls to the simulation run-time library. TYPEII includes **Cdelay, CwaitSem, CrecvMsg**, and **CreqServer**. These simulation constructs could potentially advance the simulation clock or reschedule events. Therefore, the interpretation of TYPEII is different from the interpretation of TYPEI.

4.2.1 Simulation Constructs in the Sequential Block

A simulation construct can be in any part of the model program. As shown in Figure 3, a set of sequential statements contain TYPEI simulation constructs, TYPEII simulation constructs, and C++ statements. TYPEI simulation constructs are directly translated to simulation engine library

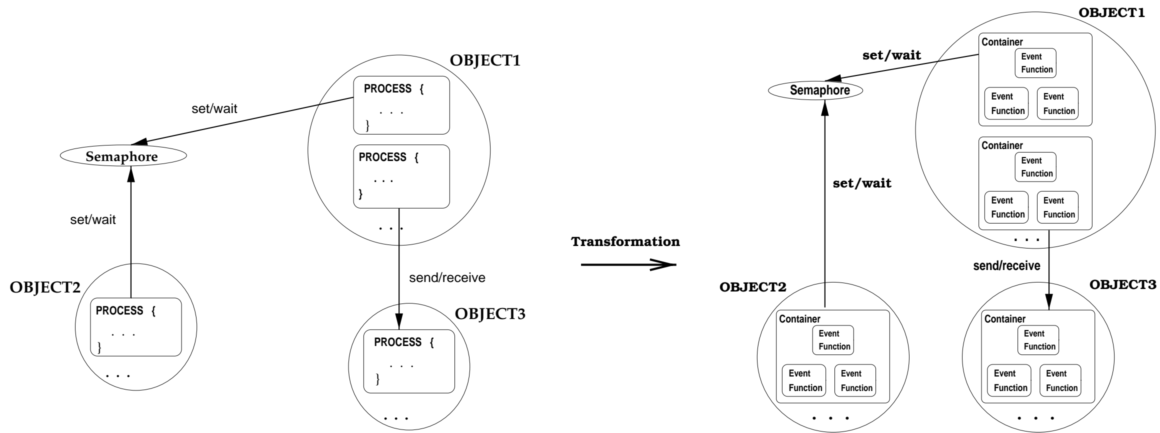

Figure 2: Model Transformation

calls. C++ statements are directly put into the event function. A TYPEII simulation construct generates an event function, since the construct influences scheduling.

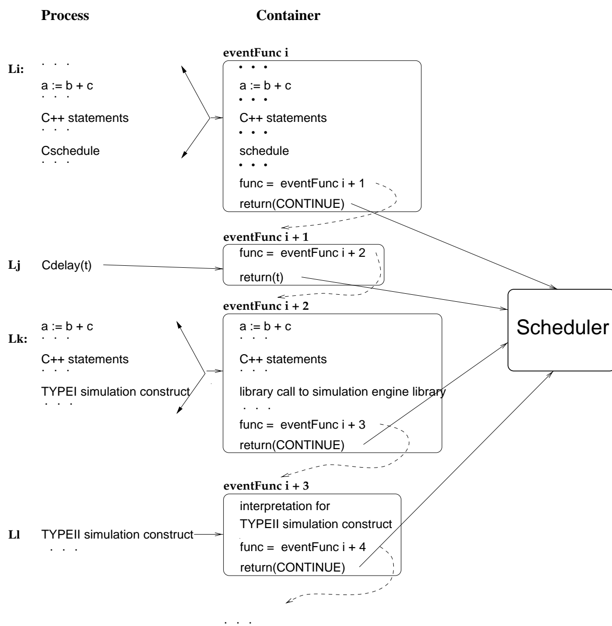

Figure 3: Simulation Constructs Interpretation in Sequential Statements

Statements between two TYPEII simulation constructs are transformed into an event function. The statements before the first TYPEII simulation construct in the code segment are also transformed into an event function, as are the statements after the last TYPEII simulation construct in the code segment.

For example, as shown in Figure 3, the TYPEI simulation construct **Cschedule** is translated to a simulation library call **schedule** in the event function "eventFunc i". **Cdelay** on line Lj results in an event function "eventFunc i + 1". Simulation time delay is returned from the event function to advance the simulation time clock. The event function "eventFunc i + 2" executes after the simulation time delay.

Two statements are added for each event function in addition to the translation for each statement of the original process. One statement sets up the invocation of the event function to execute next, and the other returns the control to the scheduler after the execution of the current event function. For example, the statement "func = eventFunc i + 1" in the event function "eventFunc + i" sets up the event function "eventFunc i + 1" to execute next. The statement, "return(CONTINUE)" returns control to the scheduler after the execution of "eventFunc i". This execution flow is based on the execution flow of the original process semantics.

Figure 4 shows the translation of **CwaitSem**, **Crecv-Msg**, and **CreqServer**. Their translation is different from that of **Cdelay** in that a "if" statement is generated. The conditional part of the "if" statement contains a call to the simulation engine library.

4.2.2 Simulation Construct inside the Conditional Block

If a TYPEII construct is in the body of a conditional block, such as a "while" statement, two event functions are generated in addition to the event functions generated as described in Section 4.2.1. This is done to retain the actions under true or false conditions.

If a TYPEII simulation construct is in the body of a control block, two event functions need to be generated (i.e., "eventFunc i + 1" and "eventFunc i + k" in Figure 5) to retain the control flow of the control flow statement (i.e., the "while" statement in Figure 5). "EventFunc i + 1" interprets the "while" statement condition and the

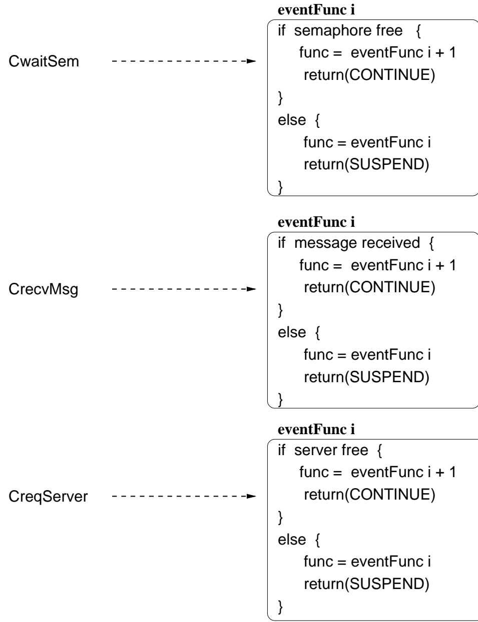

Figure 4: **CwaitSem, CrecvMsg, CreqServer** Interpretation

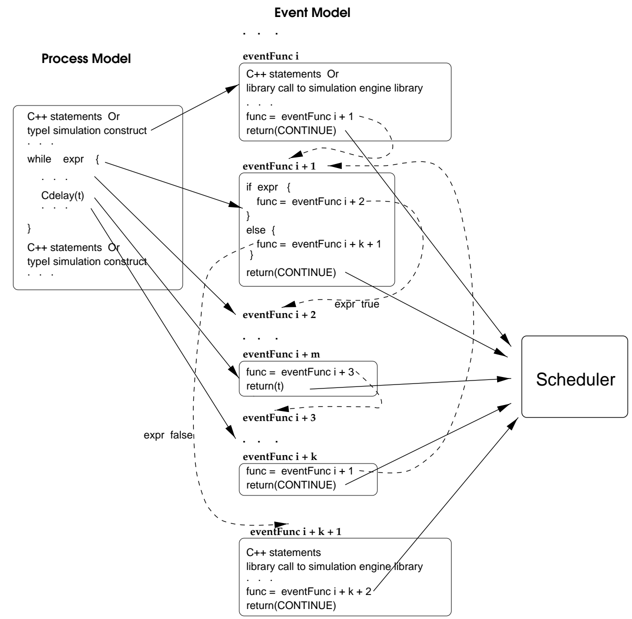

Figure 5: Simulation Constructs Interpretation in Conditional Block

tasks that need to be performed based on the condition. "EventFunc i + k" terminates the "while" statement body and jumps back to "eventFunc i + 1".

A similar transformation applies to the "if-else" block if there are TYPEII simulation constructs in its conditional statements. In this case, an event function is generated to start the "if-else" block.

5 EXPERIMENTAL RESULTS

Three examples are used to demonstrate the compiler-assisted model transformation and event-oriented simulation. The first model is an M/M/1 queue. The second one is for voting behavior of a triple-module redundant(TMR) architecture. The third one is for CSMA/CD (Carrier sense multiple access with collision detection) protocol over fast ethernet. All three examples are simulated using the proposed compiler-assisted event-oriented simulation environment and DEPEND, a process-oriented simulation package.

5.1 M/M/1 Queue

M/M/1 queue is a single-queue, single-server model with exponential interarrival and service time. As shown in Figure 6, the model is described using two objects: one for the job arrivals and the other for the service of the server. The resulting event model after transformation is shown in Figure 7 and Figure 8. For comparison, M/M/1 queue is also modeled using DEPEND.

```
Cserver* server;

class SERVICE {
    PROCESS void task() {
            server->CreqServer();
            Cdelay(expntl(interService));
            server->CendServer();
    };
};

SERVICE service[NUM];

class ARRIVAL {
    PROCESS void task() {
            i = 1;
            while (i < NUM) {
                Cdelay(expntl(interArrival));
                Cschedule(service[i], ''SERVICE::task'', currentSimTime);
                i = i + 1;
            };
    };
};
```

Figure 6: M/M/1 Queue Process-oriented Model

In Figure 9, the X-axis represents the number of jobs arrived, and the Y-axis represents the simulation time. Figure 9 shows clearly that simulation time increases linearly with the increase in the total number of jobs. Overall, DEPEND takes longer than the proposed tool. In fact, DEPEND took 1.72 seconds to simulate 2000 jobs, while the proposed tool took 0.72 seconds. When the number of jobs is increased to 4000, DEPEND took 3.4 seconds and the proposed tool took 1.36 seconds. Thus, the simulation time difference between DEPEND and the proposed tool increases as the number of jobs increases. This phenomenon is observed on both SUN4 and Ultra-sparc workstations.

```
simServer* server;

class SERVICE {
public:

        class taskER : public SimEventRoutines {
        public:
                taskER(SERVICE* ptr)
                        { erptr = ptr; func = (ER_FUNC) &taskER::task1; }

        private:
                SERVICE* erptr;
                double task1();
                double task2();
                double task3();
        };
};

/* Definitions for Object: SERVICE */

double SERVICE::taskER::task1() {
    if (server->request((SimEventRoutines* ) this)) {
        func = (ER_FUNC) &SERVICE::taskER::task2;
        return(CONTINUE);
    }
    else {
        func = (ER_FUNC) &SERVICE::taskER::task1;
        return(SUSPEND);
    }
}

//----------------------------------------------------
double SERVICE::taskER::task2() {
    func = (ER_FUNC) &SERVICE::taskER::task3;
    return(expntl(interService));
}

//----------------------------------------------------
double SERVICE::taskER::task3() {
    server->finish();
    return(EXIT);
}

SERVICE service[NUM];
```

Figure 7: Translated Code for SERVICE Object

```
class ARRIVAL {
public:
        class taskER : public SimEventRoutines {
        public:
                taskER(ARRIVAL* ptr)
                     { erptr=3Dptr; func =3D (ER_FUNC) &taskER :: task1; }
        private:
                ARRIVAL* erptr;
                double task1();
                double task2();
                double task3();
                double task4();
        } *task_Record;
};

/* Definitions for Object: ARRIVAL */
double ARRIVAL::taskER::task1() {
    i =3D 1;
    func =3D (ER_FUNC) &ARRIVAL::taskER::task2;
    return(CONTINUE);
}

//----------------------------------------------------
double ARRIVAL::taskER::task2() {
    if (i < NUM) {
        func =3D (ER_FUNC) &ARRIVAL::taskER::task3;
        return(CONTINUE);
    }
    else return(EXIT);
}

//----------------------------------------------------
double ARRIVAL::taskER::task3() {
    func =3D (ER_FUNC) &ARRIVAL::taskER::task4;
    return(expntl(interArrival));
}

//----------------------------------------------------
double ARRIVAL::taskER::task4() {
    schedule(new SERVICE::taskER(service[i]), currentSimTime);
    i =3D i + 1;
    func =3D (ER_FUNC) &ARRIVAL::taskER::task2;
    return(CONTINUE);
}
```

Figure 8: Translated Code for ARRIVAL Object

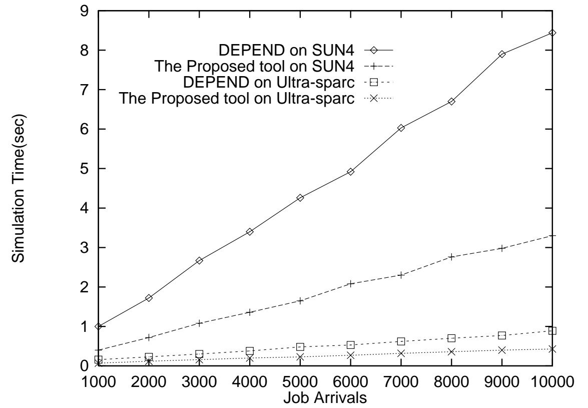

Figure 9: Simulation Time of M/M/1 Queue Example

5.2 TMR Architecture

The second model is constructed to simulate instruction execution and voting in a triple-modular redundant (TMR) architecture. Its timing diagram is shown in Figure 10. Three identical CPU modules (**P1**, **P2**, and **P3**) execute the

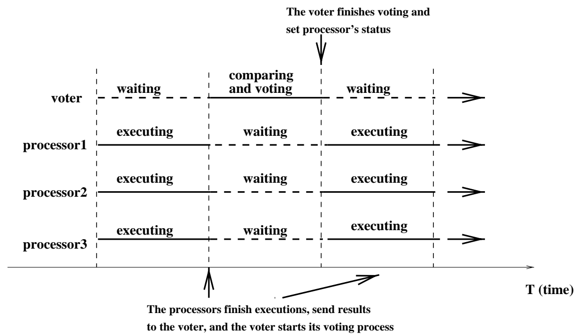

Figure 10: TMR Timing Diagram

same instruction stream synchronously. Voting is performed after finishing each instruction execution. Every CPU submits its result to the voter (**VOTER**), and suspends itself during the voting. The CPUs resume instruction execution after **VOTER** finishes voting. To simplify the simulation, only the elapse of instruction execution time is simulated.

There are communications between **VOTER** and the three CPUs to synchronize the voting and execution. The model uses semaphore and message sending/receiving simulation constructs to model necessary communication. A DEPEND model is also built and simulated. The simulation times consumed using SUN4 and Ultra-sparc are shown in Table 2. As shown in the table, the simulation time improvements of the proposed tool over DEPEND are 3.5 times on SUN4 and 4.5 times on Ultra-sparc.

Table 2: TMR Architecture Simulation Time (sec)

Simulation Package	SUN4	Ultra-sparc
DEPEND	25.11	4.55
The Proposed Tool	7.63	1.01

5.3 CSMA/CD Protocol

The system considered here consists of a group of machines interconnected using a 100Mbps high-speed ethernet. The data communication among the machines is based on the CSMA/CD protocol. According to the CSMA/CD protocol (Halsall 1992), a packet frame can be sent out when the carrier sense signal is on and the transmission medium is free. If more than one source sends frames to the medium at the same time, a collision occurs and each source must wait for a period of time based on a binary exponential distribution before a retry. Thus, more collisions mean longer waits. The flow of the CSMA/CD protocol is shown in Figure 11. The frame size is fixed, and the message size is determined randomly using a uniform distribution. Message interarrival time is assumed to be exponentially distributed.

Three objects are used to model the protocol: one ethernet object, one sending/receiving object, and one initial simulation object. The ethernet object describes the state of the ethernet. The sending/receiving object models the sending and receiving processes based on the CSMA/CD protocol. The initial simulation object initiates the simulation and all the simulation parameters. Simulation is performed to evaluate message delay from the time a message is sent to the time it is received.

Figure 12 shows the simulation time results using the proposed tool and DEPEND. In Figure 12, the X-axis represents the number of sending processes and the Y-axis represents the simulation time. Figure 12 shows clearly that the simulation time increases nonlinearly as the number of sending processes increases. Furthermore, DEPEND took much longer than the proposed tool. The simulation time difference between using DEPEND and the proposed model also increases as the number of sending processes increases.

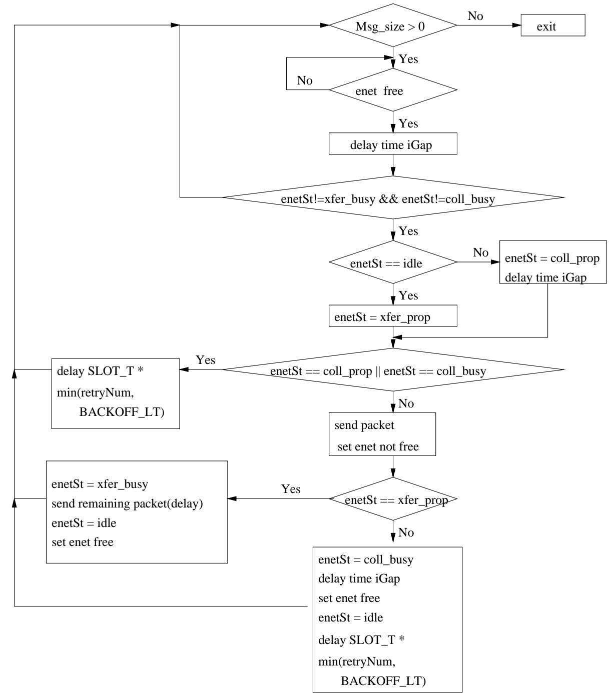

Figure 11: CSMA/CD Protocol Model

6 CONCLUSION

In this paper, an object-oriented simulation environment is proposed to speed up process-oriented simulation. The simulation environment includes an extended C++ process-oriented modeling interface, an object-oriented compiler that transforms a process-oriented model into an event-oriented model that requires no context-switching at run-time, and an event-driven simulation engine. Three simulation models were simulated within the proposed environment as well as in DEPEND and the results compared.

The proposed simulation environment offers several advantages over the current process-oriented simulation packages beyond speedup. First, no architecture-dependent context-switching or thread support is necessary, so portability becomes less of a problem. Second, the compiler performs transformation from a process-oriented model into an event-oriented model that runs on the underlying event-oriented simulation engine. If the extended C++ simulation modeling interface requires more process-level simulation constructs or if it switches to a different set of constructs, the underlying event-oriented simulation engine remains the same; only the transformation scheme needs to be adjusted based on the simulation constructs. This offers flexibility when more functionality of the modeling interface is necessary. Finally, optimization to allow more aggressive scheduling of compiler generated event functions is possible within the compiler environment.

There are certain limitations in the current implementation of the simulation environment. Currently, the extended C++ simulation modeling interface provides preliminary simulation constructs to describe simulation models. The **PROCESS** member function is the only function that can be used to specify simulation actions via simulation constructs, since only this member function is transformed

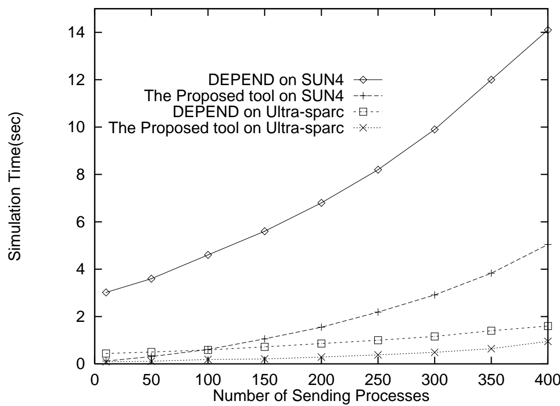

Figure 12: Simulation Time of CSMA/CD Protocol

by the compiler. Two directions for future research are possible. One is to improve the transformation techniques of the compiler to allow better modeling interface, and the other is to develop compiler optimization techniques for the generated event functions. These will help put the simulation environment into practical use, and offer more performance improvement over the context-switching based process-oriented simulation.

ACKNOWLEDGMENTS

This research was supported in part by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract DABT63-94-C-0045. The content of this paper does not necessarily reflect the position or policy of the US government, and no official endorsement should be inferred.

REFERENCES

Cooper, Eric C., and Richard P. Draves. 1988. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University.

Franta, W. R. 1977. *The Process View of Simulation*. North Holland, New York.

Goswami, K. K. 1993. Design for dependability: A simulation-based approach. Ph.D. thesis, Computer Science Department, University of Illinois, Urbana-Champaign, IL.

Jain, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. New York: John Wiley and Sons.

Jump, J. Robert. 1993. YACSIM Reference Manual. RICE University, Electrical and Computer Engineering Department.

Law, A. M., and W. D. Kelton. 1991. *Simulation Modeling and Analysis*. New York: McGraw-Hill.

MacDougall, M. H. 1987. *Simulating Computer Systems, Techniques and Tools*. The MIT Press.

Schwetman, Herb. 1986. Csim: A c-based, process-oriented simulation language. *Proceedings of the 1986 Winter Simulation Conference*, 387–396.

University of Minnesota. SIMEX.

AUTHOR BIOGRAPHIES

YIQING HUANG is a Ph.D candidate in the Computer Science Department and conducts research at the Center for Reliable and High Performance Computing at the University of Illinois at Urbana-Champaign. She received a B.S. degree in Computer Science from Tsinghua University, China and an M.S. degree in Computer Science from State University of New York at Stony Brook.

RAVISHANKAR K. IYER holds a joint appointment as Professor in the Departments of Electrical and Computer Engineering, Computer Science, and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He is also Co-Director of the Center for Reliable and High-Performance Computing. Professor Iyer's research interests are in the area of reliable computing, measurement and evaluation, and automated design.

Prof. Iyer is an IEEE Computer Society Distinguished Visitor, an Associate Fellow of the American Institute for Aeronautics and Astronautics (AIAA), and a Fellow of the IEEE. In 1991, he received the Senior Humboldt Foundation Award for excellence in research and teaching. In 1993, he received the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems."