

SimPOL: Language and Translator for Object-Oriented Process Simulation

Brian A. Malloy, Mary Jean Harrold and John D. McGregor
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

Abstract

We present our new system for object-oriented process simulation, called `SimPOL`, that consists of two main components. The first component is the `SimPOL language` that incorporates simulation constructs into C++, an existing general-purpose object-oriented language, to produce a new language that is an extension of C++. The `SimPOL language` permits the user to construct a program for simulation modeling using simulation primitives like those in `SIMULA`. The second component is the `SimPOL translator` that translates the `SimPOL language` program into a C++ program by transforming the actions of each entity in the model into a process. A base class from which each of the simulation classes inherits, provides the simulation primitives to the user's classes. We have implemented a prototype of the `SimPOL translator` that demonstrates the usefulness and portability of `SimPOL`. The main benefit of our approach is the ease of simulation programming that it provides to the user because the simulation primitives required to model processes can be learned quickly by any C++ programmer. Another important benefit of our approach over previous approaches is that our preprocessor can be used with any existing C++ compiler and is therefore, truly portable. Since C++ is the most widely used object-oriented language, `SimPOL` will be of use to a large portion of the modeling community.

Keywords: C++, modeling, object-oriented design, process-oriented simulation.

1 Introduction

Simulation is an important application of computer science that uses a model to mimic the behavior of a system. For example a factory floor assembly line may be simulated to calculate the mean cycle time of producing a workpiece, or a medical office may be simulated to determine the average time that a patient must wait to see a doctor. One way to facilitate simulation programming is to use *special-purpose* simulation languages. Although many special-purpose simulation languages, such as `SIMULA`[12, 18], `GPSS`[28], `SIMSCRIPT`[13] and `SLAM`[26], have been developed, none has emerged as *the* language for simulation modeling. There are a number of possible reasons for this: the expense of compilers for these special-purpose languages, the need to integrate the simulation model into a system written in a general-purpose language, and the resistance of programmers to learn a new language of limited applicability. Thus, *general-purpose* languages may be more appropriate for simulation programming[4, 6] as long as the underlying system can be modeled with the language.

We use an alternate approach to modeling a system that exploits the “generality” provided by a general-purpose language by extending an existing language to include simulation facilities. We have developed a new system, called `SimPOL`¹, for object-oriented process simulation that consists of two main components. The first component is the `SimPOL language` that incorporates simulation constructs into C++, an existing,

¹`SimPOL` is pronounced *simple*.

general-purpose object-oriented language, to produce a new language that is an extension of C++. The **SimPOL** language permits the user to construct a simulation model using simulation primitives like those in **SIMULA**. The second component is the **SimPOL translator** that translates the **SimPOL** language program into a C++ program by transforming the actions of each entity in the model into a process. A base class, from which each of the simulation classes inherits, provides the simulation primitives to the user's classes. We have implemented a prototype of the **SimPOL** translator that demonstrates the usefulness and portability of **SimPOL**.

SimPOL has several important advantages over existing process-oriented simulation systems. First, **SimPOL** is completely portable since we translate a **SimPOL** language program into a C++ program that can be compiled by any existing C++ compiler. By translating **SimPOL** programs into C++ programs, we obviate the need for either modifications to an existing compiler or the purchase of a special compiler for simulation modeling. Second, we exploit the object-oriented features of inheritance and dynamic binding in the design of **SimPOL**. Inheritance permits us to encapsulate the simulation activities and hide them from the user, while incorporating them into each simulation object. Through dynamic binding, we pass control among objects in the simulation in an extensible fashion, so that new types of processes can be added to the model without modifying existing objects. Our implementation of processes permits efficient management of memory during execution since the overhead due to function calls is kept to a minimum. Finally, **SimPOL** provides ease of simulation programming because the simulation primitives required to model processes can be learned quickly by any C++ programmer. Since C++ is the most widely used and quickly growing object-oriented language, **SimPOL** will facilitate object-oriented design for a large portion of the modeling community.

In the next section, we give details of the **SimPOL** system by discussing both the language and the translator. In section 3, we discuss our prototype implementation. Section 4 presents related work and compares it to the **SimPOL** system, and section 5 gives our conclusions and future directions.

2 **SimPOL: Language and Translator**

SimPOL, depicted in Figure 1, consists of two main components: the **SimPOL** language and the **SimPOL** translator. The **SimPOL** language extends C++ with simulation primitives, and the **SimPOL** translator transforms a **SimPOL** program into a C++ program. In the next two sections, we detail the language and the translator.

2.1 **The SimPOL Language**

SimPOL provides functionality to facilitate process-oriented simulation modeling. A process-oriented approach to modeling requires a language that includes a construct known as a process. A *process*, as defined by **SIMULA**, is a retentive structure whose associated control point and storage is retained even when the locus of control no longer resides in the body of the retentive structure. **SimPOL** provides language facilities to support creation and manipulation of processes that are essentially the same as those of **SIMULA**. A process in **SimPOL** is defined in a simulation class where a *member function* in the class contains all of the operations of the process, and the *data members* include local data and parameters together with information to restore control upon re-entry. A process is created in **SimPOL** using a **new** function call, which is similar to the process creation facility of **SIMULA**. This **new** function call has the same syntax and semantics as the **new**

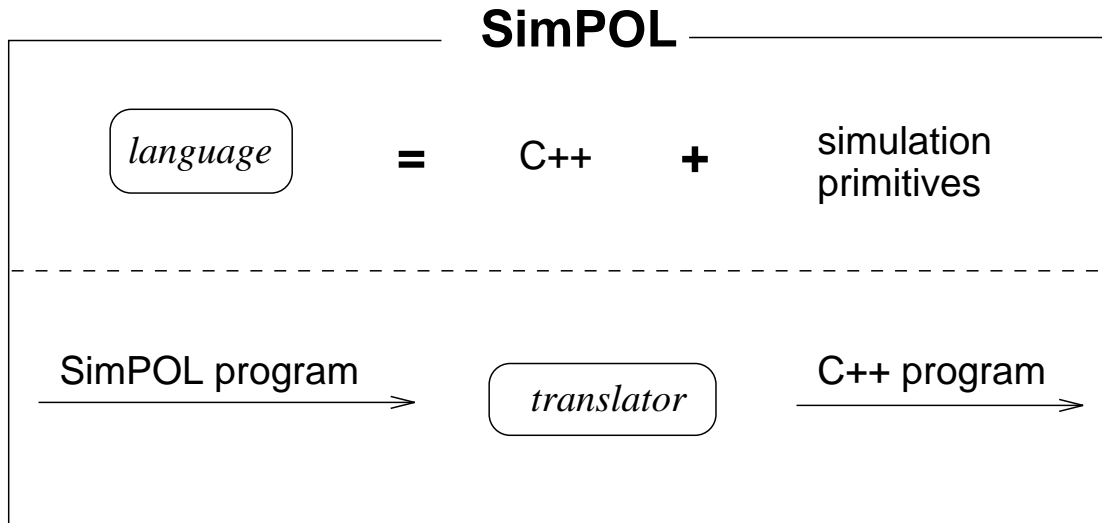


Figure 1: **SimPOL** consists of a language that extends C++, and a translator that converts a **SimPOL** program to C++.

function call in C++. Process operations in **SimPOL** are manipulated with the following simulation primitives:

- Activate()**: schedule a process for execution
- Passivate()**: suspend execution of the active process
- Hold(T)**: suspend execution of the active process for time T
- ActivateAt(T)**: schedule a process for execution at time T
- Time()**: return system time

A process control point is specified by a process operation applied either to the process itself or to another process through the use of the simulation primitives. Figure 2 illustrates process synchronization using some of these simulation primitives. In the figure, execution begins in **Process A** but is transferred to **Process B** by the first **B -> Activate()** primitive. After the execution of some statements in **B**, **Process B** suspends itself using the **Passivate()** primitive, and therefore, becomes unscheduled. Execution is then transferred to the previously active process, **Process A**, where execution continues until the next primitive, **B -> Activate()**, is encountered. Again, control transfers to **Process B** and continues until the next primitive, **Hold(2)**. Execution resumes to completion in **Process A** after which control is transferred automatically to **Process B**. **Process B** resumes execution after **Hold(2)** with system time incremented by two units. **Process B** then executes to completion.

SimPOL allows for control to be passed among processes both implicitly and explicitly. *Implicit* control is achieved through the event list, **Event List**, which keeps an ordered list of processes to be executed; processes are ordered by time on **Event List**. When the currently active process relinquishes control without explicitly naming the next process (i.e., **Passivate()**, **Hold()**), control passes automatically to the process at the head of **Event List**. In the example in Figure 2, when **Hold(2)** is executed, **Process B** must be reinserted into **Event List** with a “time stamp” value equal to the present value of system time plus two

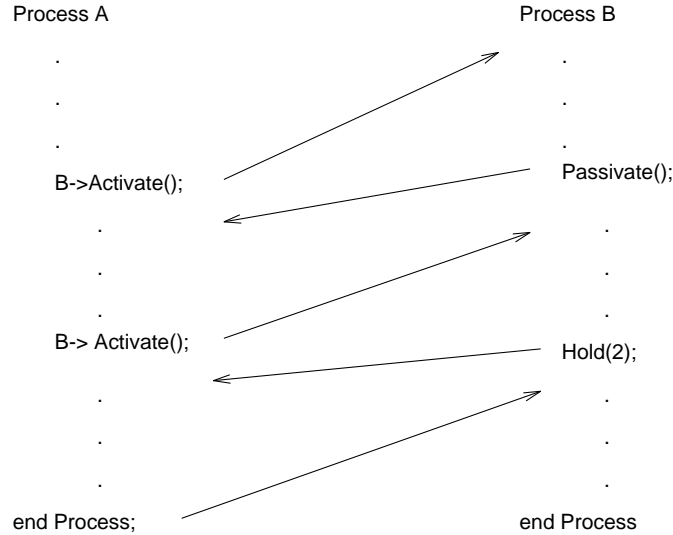


Figure 2: Process synchronization using **SimPOL** primitives.

units. Since **Process B** is suspended by the **Hold()** primitive, execution reverts to the previously active process, **Process A**, through **Event List**. When **Process A** finally terminates, because all of its statements have been executed, it becomes unscheduled and control reverts to the process at the head of **Event List**.

Through the simulation primitive $p \rightarrow \mathbf{Activate}()$, control is passed *explicitly* to p , the next process to become active. Explicit transfer of control is shown in Figure 2 by the $\mathbf{B} \rightarrow \mathbf{Activate}()$ primitive. The $\mathbf{B} \rightarrow \mathbf{Activate}()$ primitive places **Process B** at the head of **Event List**, and therefore, **B** becomes the active process, with system time unchanged.

The simulation primitives, along with other simulation data and activities, are encapsulated in our abstract base class, **Sim_Process**. Programmers define a simulation class by inheriting from **Sim_Process**. Each simulation class represents an entity in the simulation model. All actions of that entity that use the simulation primitives must be included in a special member function called **Script()**, contained in the simulation class.

Modelers require list facilities for inserting processes into, or removing processes from, user-defined lists; list facilities are included in the **SIMULA** language in a special class called **SIMSET**. Using the inheritance feature of object-oriented languages, **SimPOL** users can use their own list facilities available in a C++ class library for insertion or removal of processes.

To illustrate the usage of **SimPOL**, we now discuss a process-oriented model of a factory floor simulation. In the model, an assembly line is simulated to calculate the mean cycle time required for a workpiece to become a finished product. Workpieces enter the assembly line at a constant rate and are added to a drill queue so that when a drill is free, one of the workpieces is operated upon first by that drill. After being drilled, a workpiece enters a thread queue so that when a lathe is free, the workpiece is threaded by that lathe. Finally, a workpiece enters an inspection queue so that when an inspector is free, the workpiece is examined before it leaves the factory as a finished product. During the simulation, after a lathe operates on a constant number of workpieces, it must be adjusted; the lathe is inactive during this adjustment period.

```

simQueue *drillQ, *threadQ, *inspectionQ;
simQueue *drills, *lathes, *inspectors;

Drill::Script() {
    while(1) {
        if (!drillQ -> isEmpty() )
            w = drillQ -> get();
        else {
            drills -> put(this);
            Passivate();
            w = drillQ -> get();
        }
        Hold(DRILL_TIME);
        w -> Activate();
    }
}

main() {
    // see Appendix A for declarations
    // see Appendix A for initializations

    i = 0;
    while (i <= totalSimulationTime) {
        i++;
        w = new WorkPiece(i);
        w -> Activate();
        Hold( holdTime );           // holdTime input by function inputParameters()
    }
    Hold(100000);
    printTotals();
}

```

Figure 3: Two functions, coded in **SimPOL**, for a factory floor simulation; the other important classes and member functions for this program are given in Appendix A.

Figure 3 shows two **SimPOL** functions from a factory floor simulation; the other important classes and member functions for this simulation program are given in Appendix A. To simplify our presentation, we omitted statements that would report simulation output to the user; these output actions are summarized in a call to function `printTotals()`. Figure 3 shows only function `main()` and the process `Drill`; in the complete simulation model there are four classes (processes) that are used to represent lathes, drills, inspectors and workpieces.

In `main()` in the **SimPOL** program, instances of drills, lathes and inspectors are created during initialization using the `new` function. Then, in the `while` loop in `main()`, a `WorkPiece`, `w`, is created using the `new` function, and activated using the `Activate()` simulation primitive. This action causes this instance of a `WorkPiece` process to be placed at the head of `Event List`, and execution continues by performing the actions of the `WorkPiece`. When control returns to `main()`, it will `Hold()` for a constant number of cycles `holdTime`, to represent the constant arrival rate of a workpiece. Workpieces are created in the `while` loop of `main()` for `totalSimulationTime` cycles. After the `while` loop terminates, `main()` executes a `Hold(100000)` to allow any workpieces in queues to finish. Finally, the output of the simulation is reported

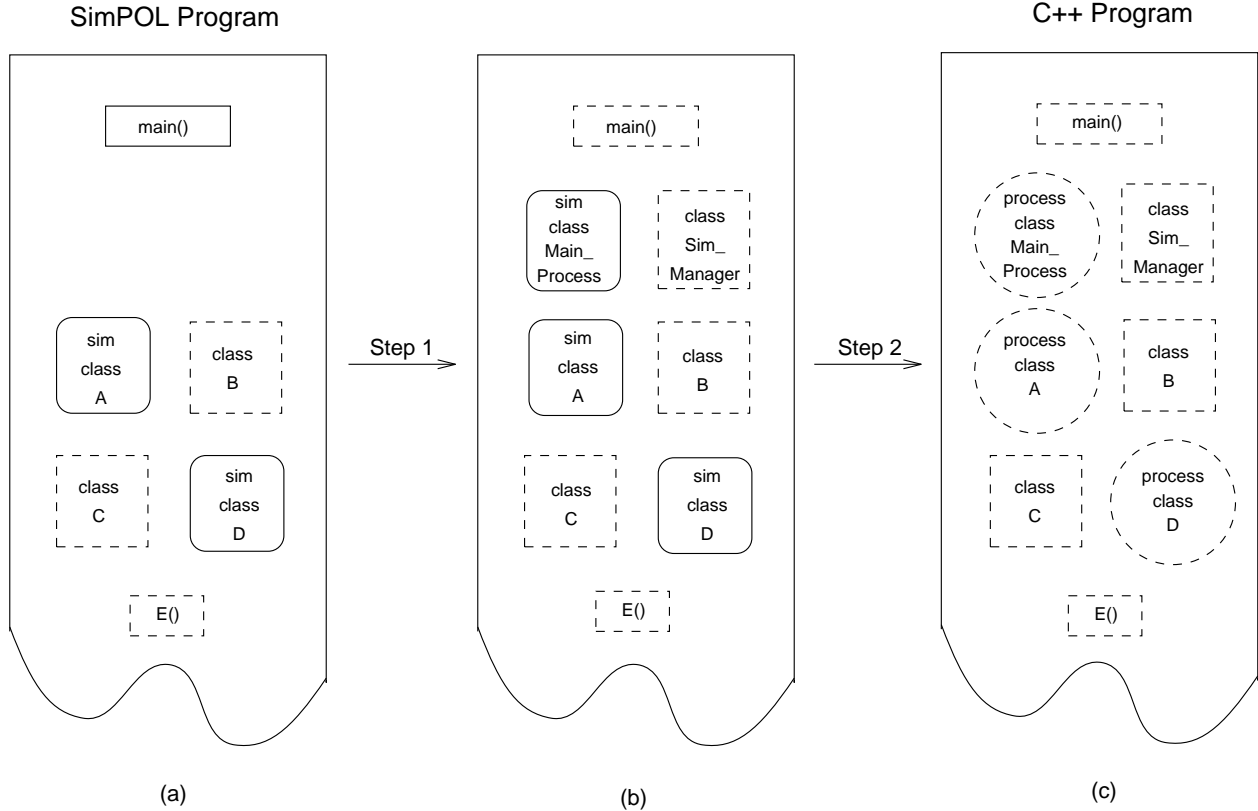


Figure 4: Overview of the translation from **SimPOL** to **C++**. Figure (a) represents a **SimPOL** program, figure (b) represents its partial translation, and figure (c) represents the resulting **C++** program.

and the simulation terminates.

2.2 The **SimPOL** Translator

The **SimPOL** translator accepts a **SimPOL** program, and translates it into a **C++** program. Figure 4 illustrates the main steps in the translation of a **SimPOL** program. At each phase of the translation, rectangles represent functions, squares represent non-simulation classes, rounded squares represent simulation classes, and circles represent process classes. Components in the figure that require further translation are shown as solid figures, and components that require no further translation are shown as dashed figures. The illustrated **SimPOL** program in Figure 4(a) has functions `main()` and `E()`, non-simulation classes `B` and `C`, and simulation classes `A` and `D`; function `main()` and simulation classes `A` and `D` require translation.

In Step 1 of the translation, we incorporate simulation management into the program by inserting a simulation manager class, `Sim_Manager`, converting the program's `main()` into simulation class `Main_Process`, and creating a new `main()` for the program. the `Sim_Manager` class handles the simulation by activating processes when execution begins. The program's original `main()` is converted to a simulation class so that it can be translated to a process class in Step 2. The new `main()` instantiates the old `main()`'s process class and calls `Sim_Manager`. Figure 4(b) illustrates the results of Step 1 of the translation: class `Sim_Manager` is

added, `main()` is now a simulation class `Main_Process`, and there is a new `main()` for the program. Only simulation classes (i.e., `Main_Process`, `A` and `D`) require further translation.

In Step 2 of the translation, we convert all simulation classes to process classes using our partitioning algorithm. Classes that are defined in the user's program, but do not use simulation primitives (e.g., class `B`), are not examined by the translator; this improves the efficiency of the translation process. We utilize the object-oriented features of C++ during our translation process. All simulation classes created by the user are unified under a single parent class using inheritance to provide type compatibility for the list manipulation, and to provide many of the overhead features required for the simulation. Our management of the actions of the processes prevents `Sim_Manager` from generating a lengthy sequence of function calls and causing the runtime stack to grow excessively. We show the resulting C++ program in Figure 4(c). Simulation classes (i.e., `Main_Process`, `A` and `D`) have been translated into process classes.

We detail the steps in the translation from `SimPOL` to C++ in the following sections.

2.2.1 Step 1: Inserting simulation management

Inserting simulation management into a `SimPOL` program requires inserting simulation class `Sim_Manager`, converting `main()` to a simulation class and creating a new `main()`.

The simulation manager, class `Sim_Manager`, is added to the program. `Sim_Manager` performs three important actions. First, `Sim_Manager` handles the scheduling and unscheduling of processes. Processes are scheduled by inserting them into `Event List` according to their "time stamp". Processes are unscheduled by removing them from `Event List`. Second, `Sim_Manager` directs the progress of the simulation by updating global simulation time to reflect the "time stamp" of the currently executing process, which is at the head of `Event List`.

Finally, `Sim_Manager` handles the transfer of control to processes at the head of `Event List`. A function `Transfer()` in `Sim_Manager` accomplishes this transfer by updating global simulation time to reflect the time-stamp of the process at the head of `Event List`, and transferring control to this process by making a call to its `Script()`. The dynamic binding of C++ ensures that the appropriate `Script()` is called for the process at the head of `Event List`.

The `main()` function of the `SimPOL` program is converted to a simulation class by defining `Main_Process` as a subclass of class `Sim_Process`². `Sim_Process` contains a **virtual** member function `Script()` which will be defined in each subclass and will contain all of that class's simulation actions. The actions in `main()` are placed in the `Script()` of `Main_Process`. This `Main_Process` simulation class will be translated into a process class in Step 2.

Since function `main()` has been converted to a simulation class, a new `main`, `main()`, is created for the program; Its functionality is the same for all simulation programs. The actions of new `main()` are listed in Figure 5. In Action 1 of Figure 5, an instance of `Sim_Manager` is created, and in Action 2, an instance of `Main_Process` is created. In Action 3, the `Main_Process` object is inserted at the head of `Event List`, since it must be the first process to execute. Finally, in the fourth action, control is transferred to `Sim_Manager` by calling one of its functions `Transfer()`, which handles the transfer of control among the simulation processes.

²Recall that all simulation classes in the `SimPOL` program must be derived from `Sim_Process`.

Action 1: Create a new `Sim_Manager`;
Action 2: Create a new `Main_Process`;
Action 3: Insert instantiated `Main_Process` into `Event_List`;
Action 4: Transfer control to the instantiated `Sim_Manager`;

Figure 5: Actions required for new `main()` in the translated `SimPOL` program.

One of the important benefits of our system is that, because of our management of the actions of a process, we minimize the runtime stack overhead due to simulation. When the simulation begins, an activation for the new `main()` is placed on the runtime stack. Then the new `main()` calls `Transfer()`, and an activation for `Transfer()` is placed on the runtime stack. The activations for `main()` and `Transfer()` are the only two activations that remain on the runtime stack during the simulation. At any time during the simulation, there will only be one additional activation, due to the simulation, on the runtime stack, and that activation will represent the currently executing process.

A process A in the simulation becomes active in one of two ways: (1) another process in the simulation activates A by using a simulation primitive, or (2) process A moves to the front of the event list. When a process becomes active, control is transferred to that process by invoking the corresponding `Script()` and transferring control to the next portion of the `Script()` to execute. Control then leaves that process by returning from `Script()`; thus, the activation for the `Script()` only remains on the stack while the partition is executing. When the next process is activated, the record for that process is placed on the stack and then removed when the actions are executed.

To illustrate, consider the two snapshots of the runtime environment shown in Figure 6. In Snapshot 1, process A is currently executing since it is at the head of `Event_List`. Snapshot 2 represents the runtime stack after process A is removed from execution by one of the simulation primitives and, for this example, is placed at the end of the `Event_List`. Now there is an activation for process B on the runtime stack, but no remaining activation for A.

2.2.2 Step 2: Translating simulation classes to process classes

To incorporate the notion of a process into C++ we must address the problem of modeling a retentive structure by a recursive function. When the function terminates, information must be saved and, when the associated instance of the process is resumed, the information must be restored. The saved information includes local data and the last active control point that will be used for the next re-entry into the function. The semantics of a process can be partially simulated by a recursive function but a retentive component to maintain information that remains invariant between activations of the process is also needed.

We translate each simulation class into a process class that can be compiled by any C++ compiler. When a simulation primitive is encountered during execution, control passes from the `Script()` in the current

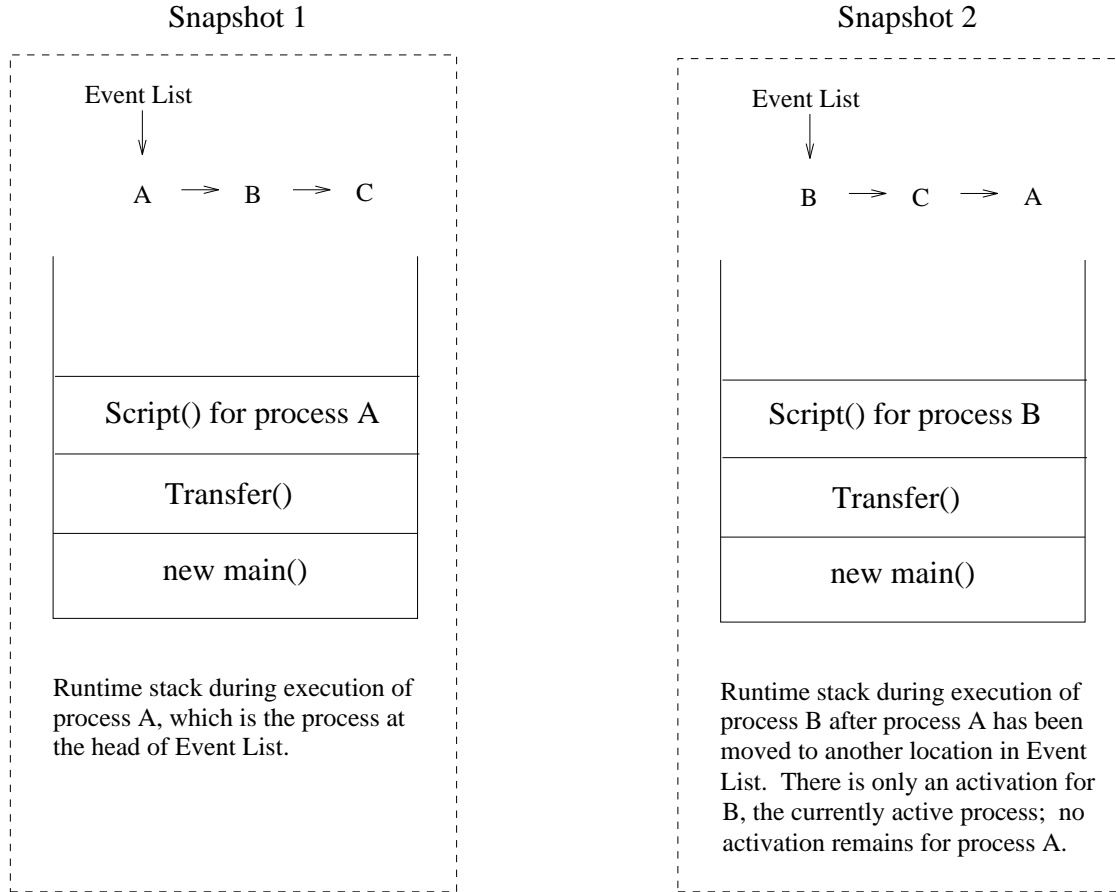


Figure 6: Two snapshots of the runtime stack for a simulation, which illustrate the low simulation overhead required by our `SimPOL` system.

object, to the `Script()` of other objects in the simulation application. If control returns to this `Script()`, execution will continue at the statement following the simulation primitive. To accomplish the translation of a simulation class to a process class, we first identify basic blocks and control points specified by simulation primitives contained in the `Script()` member function of the simulation class. We then partition the code, where each partition is associated with a basic block in the control flow graph of the `Script()`. We construct the control flow graph for the `Script()` in the usual manner except that a simulation primitive marks the end of a basic block. We then use a `switch` statement to simulate the semantics of the simulation class by inserting a partition into each option of the `switch` statement. We use a partition counter, inserted into the data member of the simulation class, to choose the appropriate partition in the `Script()` to execute.

Our algorithm `Translate` that translates a simulation class, `SC_SimPOL`, in a `SimPOL` program into a process class, `SC_C++` in a C++ program is given in Figure 7; since `SC_SimPOL` is a simulation class, it contains a `Script()` member function, that contains all accesses to simulation primitives in that class. `Translate` consists of three steps. In Step 1, a control flow graph, `CFG`, is constructed for `Script()`. The control flow

```

algorithm Translate(SC_SimpPOL):SC_C++
input      SC_SimpPOL: a simulation class, containing Script()
output    SC_C++: a C++ class
declare   CFG : control flow graph
           Bi, Bj, Bk : basic block in CFG

begin Translate
/* Step 1: Construct control flow graph for Script(); B0 is first basic block, BN is last basic block */
CFG = Construct control flow graph for Script(); construction is modified to handle simulation primitives
CFG = Transform CFG by replacing:
    each loop statement (while, for, do-while) with an equivalent if-then/goto statement
    each implicit transfer of control (continue, break) with an equivalent goto statement
Mark each basic block in CFG
/* Step 2: Enclose control flow graph with a while loop by adding new basic blocks and edges */
CFG = Enclose CFG with a new while loop by adding:
    a basic block B-1 containing "flag = 1"
    a basic block WH containing "while(flag) "
    a basic block BN containing "flag = 0" and mark it
    a basic block BN+1 representing the new end of the program
    edges (B-1,WH), (WH,B0) labeled "T", (WH,BN+1) labeled "F", and (BN,WH)
/* Step 3: Partition program to incorporate retentive control */
CFG = Translate CFG by adding
    "switch(PC)" as the first basic block SW of the while loop added in Step 2
foreach marked basic block Bi in CFG do
    if Bi contains a goto  $s_j$  statement then
        Replace this statement with "PC = k; break", where Bk is the basic block containing  $s_j$ 
    elseif Bi contains an if(condition) statement with Bj the true target and Bk the false target then
        Replace this statement with " if (condition) PC = j; else PC = k; break;"
    elseif the last statement in Bi is a simulation primitive then
        Add "PC = j" to the beginning of Bi, where Bj is the successor of Bi over a simulation edge
        Replace "(call to simulation primitive)" with "return (call to simulation primitive)"
    else for edge (Bi,Bj) in CFG
        Add "PC = j; break" to the end of Bi
    Remove all edges in CFG emanating from Bi
    Add (SW, Bi) with label i to CFG
    if Bi does not contain a simulation primitive then
        Add (Bi, WH) to CFG

end Translate

```

Figure 7: Algorithm to translate a simulation class into a process class.

graph is constructed in the usual way[2] except that

- the statement following the use of a simulation primitive begins a new basic block and
- there is no control flow edge emanating from a basic block whose last statement is a simulation primitive; instead there is a *simulation* edge connecting this block with the block where control will return, if it ever does, in this member function.

After the CFG is constructed, we replace all loop statements, such as **while**, **for** and **do-while**, by equivalent **if-then/goto** statements. We also replace implicit transfers of control, such as **continue** and **break**, with **goto** statements. By making these replacements, we facilitate our later semantics preserving translation from **SimpPOL** to C++. At this point, we mark all nodes in the CFG, since they will be considered for partitioning.

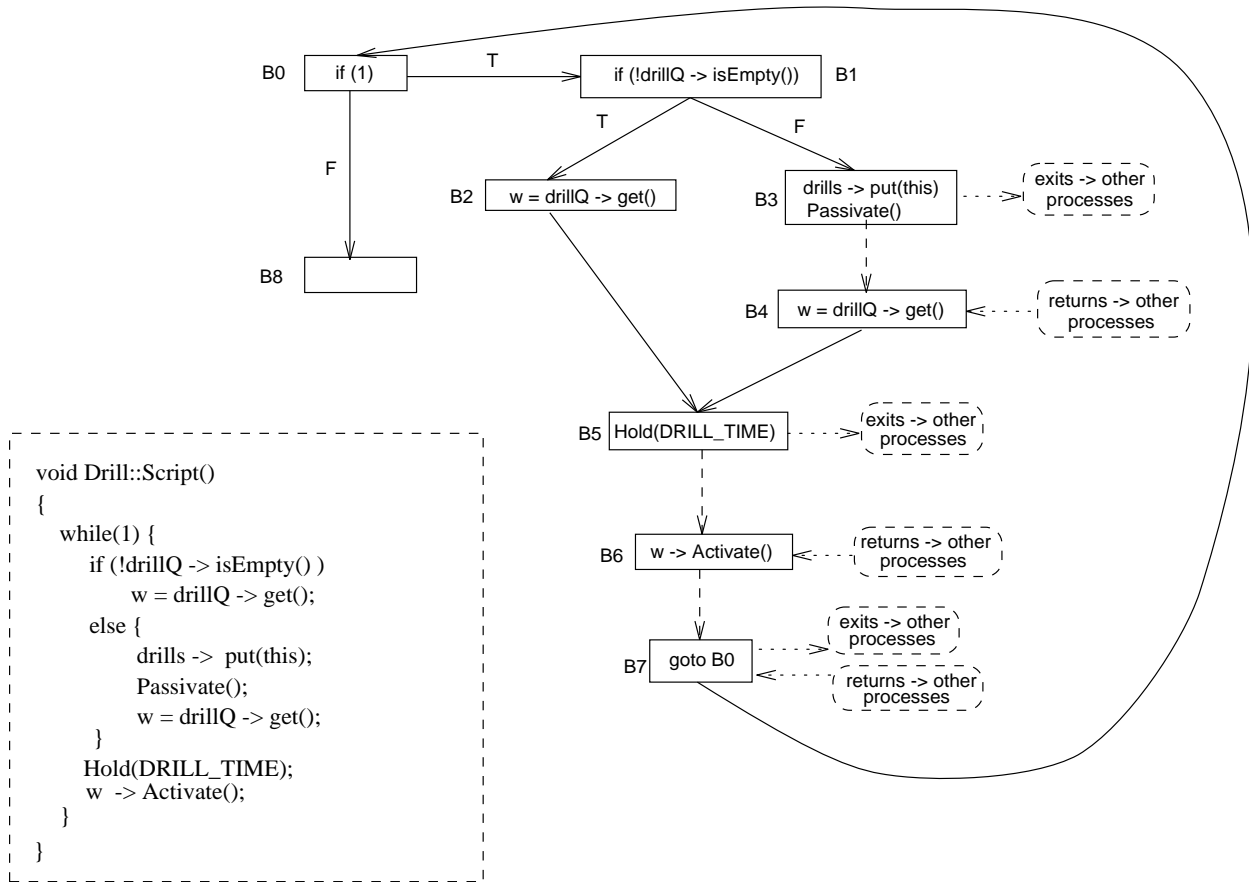


Figure 8: Code for simulation class `Drill` of Figure is shown in the dashed box. The resulting program after Step 1 of `Translate` is applied to the `Script()` member function of `Drill`.

To illustrate this initial translation, we revisit the `Drill()` class of Figure 3. Figure 8 shows the control flow graph for `Script()` in simulation class `Drill` after the completion of Step 1. Control flow edges, such as (B2,B5), are shown as solid edges; simulation edges, such as (B5,B6) are shown as dashed edges. Dotted edges in the figure model the execution of simulation primitives which cause control to leave `Script()`; if control returns to `Script()`, execution will continue at the first statement following the primitive. The `while` loop at B0 is changed to an `if-then/goto` statement by replacing “`while(1)`” with “`if (1)`”, and adding the “`goto`” statement in B7; the semantics of the original member function are preserved with this transformation. No control passes directly from B5 to B6 due to the use of the simulation primitive `Hold()` in B5. However, if control ever does return to this process, it will return at B6; a similar situation exists for B6 and B7.

In the next step in `Translate`, we enclose the code produced by Step 1 in a `while` loop that will be used to provide retentive control. To do this, we add several basic blocks and statements: a basic block, labeled B-1, containing a statement, “`flag = 1;`”, that initializes the loop control variable; a basic block, labeled WH, containing the `while` loop header, “`while (flag) {`”; a basic block, labeled BN, containing a

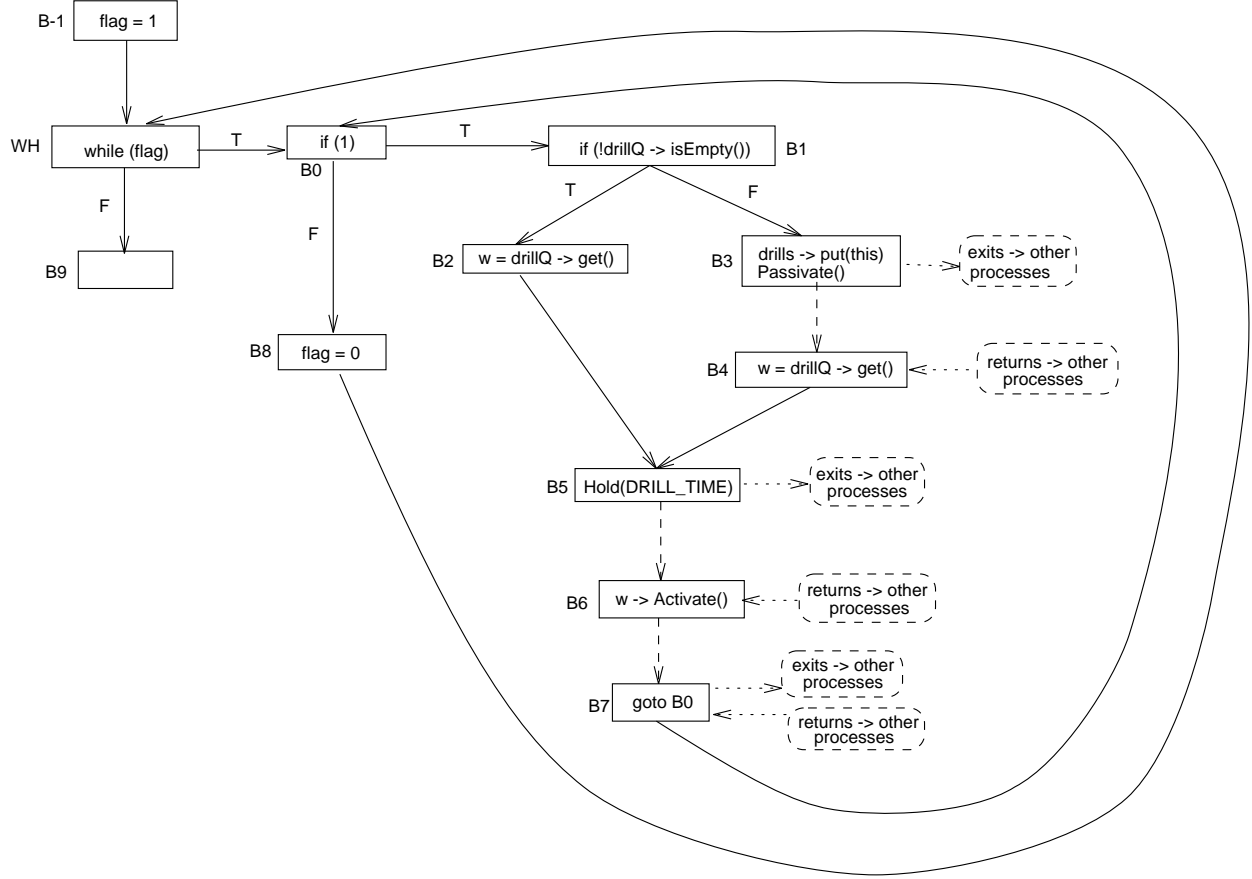


Figure 9: The resulting program after Step 2 of **Translate** is applied to the control flow graph of Figure 6.

statement, “flag = 0; }”, that changes the flag to provide exit from the loop; and a basic block, labeled BN+1, that represents the new end of the program. We also add appropriate edges representing the flow of control between the new basic blocks. Finally, the block containing “flag = 0; }” is marked. Since this added **while** loop in Step 2 only executes one time for the code in **Script()**, the code that results is semantically equivalent to the original code in **Script()**. In the next step of the translation, this **while** loop, together with an inserted **switch** statement, is used to implement retentive control.

Figure 9 shows the results of applying Step 2 of **Translate** to the control flow graph in Figure 8. The added **while** loop is entered and the code from the original program is executed until completion. At this time, “flag=0” is executed. When the **while** loop condition is tested it is false, and the loop is exited.

The final step of the translation from a **SimPOL** program to a C++ program is to use the basic blocks to partition the code. A **switch** statement is added after the **while** loop at WH to control transfer to the partitions. When the object is instantiated, a *partition counter*, PC, is initialized to 0. When the **switch** is executed, it transfers control to the partition whose number is the current value of PC. In each basic block in the control flow graph produced by Step 2, we make every transfer of control explicit. To do this, we set

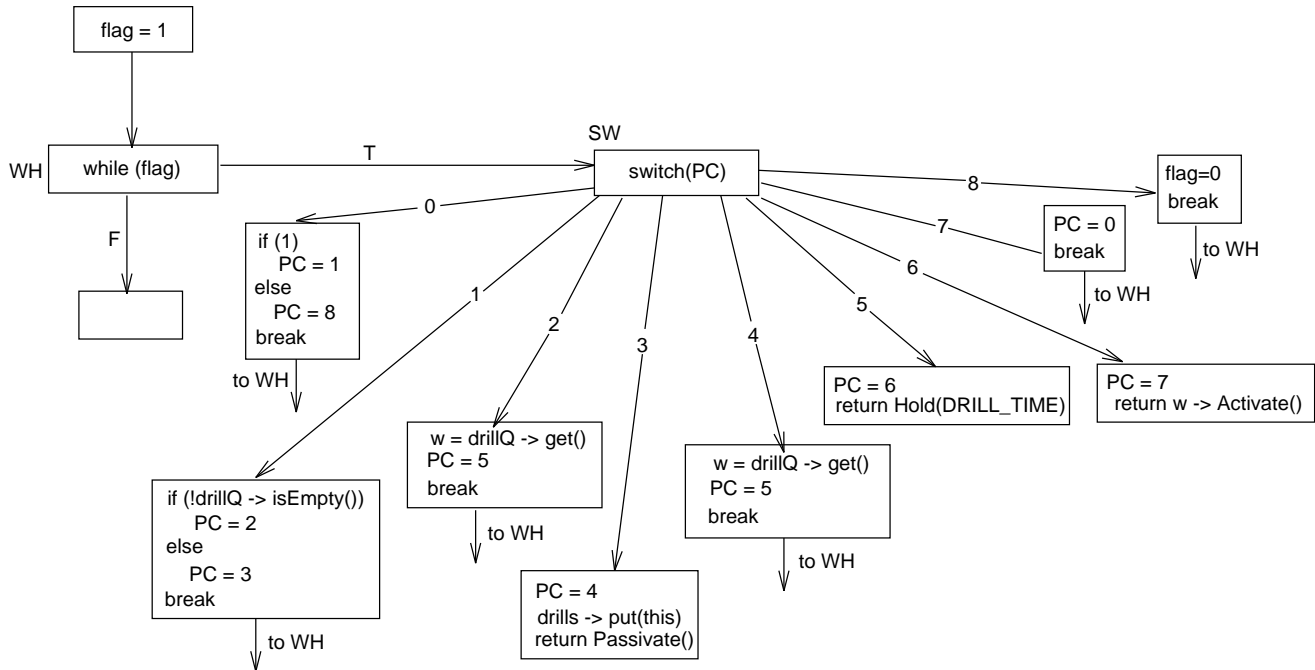


Figure 10: The resulting C++ program after Step 3 of **Translate** is applied to the control flow graph of Figure 9. Numbers along the edges of the graph correspond to basic blocks in the original control flow graph and these numbers are case options in the resulting **switch** statement shown in Figure 11.

PC to the next basic block (partition) to be executed. After the basic block is executed, control returns to the top of the **while** loop, WH, and then to a **switch** statement, which transfers control by considering the current value of PC.

To illustrate the final phase of the translation, consider the program in Figure 9 and the resulting translation of the control flow graph shown in Figure 10. The **goto** statement in B7 transfers control to B0. Thus, setting PC to 0, inserting a break statement, and replacing edge (B0,B1) with edge (B0,WH) causes control to transfer to partition (basic block) 0, which preserves the semantics of the original program. Basic block B0 contains an **if** statement. If the conditional statement is true, control is transferred to the true branch at B1. Thus, our translation sets PC to 1, inserts a break, and replaces edge (B7,B0) with edge (B7,WH). These additions and replacements cause control to transfer to WH, and finally to partition (basic block) 1, which is the true branch of the conditional statement. If the conditional statement is false, (which it never is in this example), control transfers in a similar fashion to the false branch whose PC is 8. Basic block B3 contains a simulation primitive, **Passivate()**. The execution of **Passivate()** causes control to transfer from the **Script()** in **Drill**; if transfer ever returns to **Script()**, it should return to B4. Our translator sets PC to 4, and translates the simulation primitive **Passivate()** to **return Passivate()**. At some later time, if execution returns to **Script()**, it will return to the beginning of **Script()**. However, since PC contains 4, control will transfer to partition (basic block) 4, thus implementing the retentive control required for the

```

void Drill::Script() {
    int flag, WH = -1;

    flag = 1;
    while (flag) {
        switch (PC) {
            case 0: if (1)
                    PC = 1;
                    else
                    PC = 8;
                    break;
            case 1: if (!drillQ -> isEmpty() )
                    PC = 2;
                    else
                    PC = 3;
                    break;
            case 2: w = drillQ -> get();
                    PC = 5;
                    break;
            case 3: PC = 4;
                    drills -> put(this);
                    return Passivate();
            case 4: w = drillQ -> get();
                    PC = 5;
                    break;
            case 5: PC = 6;
                    Hold(DRILL_TIME);
            case 6: PC = 7;
                    return w -> Activate();
            case 7: PC = 0;
                    break;
            case 8: flag = 0;
                    break;
        }
    }
}

```

Figure 11: Resulting C++ code for `Script()` in `Drill` after translation.

primitive. Finally, in B2, PC is set to 5, which is the partition containing the next statements to be executed. Edge (B2,B5) is replaced with edge (B2,WH). The resulting C++ code produced for the `Script()` for `Drill` is shown in Figure 11.

3 Implementation

To illustrate both the power of `SimPOL` in facilitating simulation modeling and the portability of `SimPOL` across compilers and machines, we implemented a prototype of the `SimPOL` translator and ported it to a variety of platforms. `SimPOL`'s expressive power is demonstrated by the ease with which users are able to construct intricate simulation models in a short period of time. `SimPOL`'s portability is demonstrated by its port from the PC environment, operating under MS-DOS using Borland C++[16], to the Sun workstation

environment, operating under Unix using both the AT&T[1] and the GNU C++[34] compilers.

We used our prototype to translate the **SimPOL** program for the factory floor simulation, and we compiled the resulting C++ program with the GNU C++ compiler. Using the Sun Sparc-10 workstation, translation of the **SimPOL** program required 1 second, and execution of the simulation program required 5 seconds for a model factory containing 5 drills, 7 lathes, 3 inspectors, 10 workpieces generated per hour and 10,436 workpieces generated in the system with a mean cycle time of 150 cycles. We include these timings to illustrate the potential efficiency of **SimPOL**. The resulting C++ code for the **Drill** process class is shown in Figure 11.

4 Comparison with other Simulators

We now compare **SimPOL** with related simulation languages. We focus on languages that support the process view [11] of simulation since this approach to modeling is accepted for its advantages over other approaches[8] and is the most popular approach among commercially available simulation languages[9]. We discuss both scenario languages and general purpose languages that offer the process view.

Process-oriented simulation languages can be partitioned into scenario languages and general-purpose (procedural) languages[17]. Models are created in scenario languages by specifying scenes composed of block diagrams that are activated by transactions. Examples of scenario languages are SIMOBJECT[14], ProModel[15], Extend[10], Q+[23] and WITNESS[35]. In addition to simulation primitives, general purpose languages also provide language constructs such as loops and decision structures. While scenario languages allow the modeler to quickly construct a high level simulation model, general-purpose languages give the programmer more control over the model and are more widely applicable because of the flexibility provided by the language constructs[27]. We restrict our discussion to the more widely applicable general-purpose languages.

There is a large selection of general-purpose simulation languages that facilitate the process-oriented approach to simulation modeling[5, 12, 13, 25, 27, 33]. The canonical example of a general-purpose process-oriented simulation language is SIMULA[12], which is based on the block structured language Algol[24, 36, 3]. SIMULA introduced the notion of a process as well as the concept of a class as language constructs. SIMULA uses class prefixing to allow a prefixed class to inherit data and operations from the prefix class. Thus, SIMULA is the ancestor of the process-oriented approach to modeling and the object-oriented approach to software design. Another general-purpose process-oriented language is MODSIM II[5], which is based on Modula-2[38]. The preprocessor approach is used by **SimCal**[20], to incorporate the process view of simulation into Pascal, the base language[37]. Since these approaches are based on Algol, Modula-2 and Pascal, the simulation modeler is not required to learn a new language to construct models. However, the use of SIMULA and MODSIM II requires a special purpose compiler and the resulting programs are not portable across compilers or machines. Further, The base languages for SIMULA, MODSIM II and **SimCal** are infrequently used and the design of **SimCal** does not handle unstructured programs.

Other general-purpose process-oriented languages include SIMSCRIPT[13], SLAM II[26], SIMAN[25], GPSS/H[33], and HSL[27]. While these languages offer excellent support for modeling, they require the programmer to learn a new language in order to construct simulation models. Further, the constructed models are dependent on the compiler as well as the machine that supports the compiler.

A popular general-purpose simulation language that facilitates the process view is CSIM[29, 30, 31]. CSIM is based on C and a later version, CSIM/C++[32], was extended to C++. Since CSIM is based on C and C++, the programmer familiar with these languages can code simulation models without learning a new language. However, the design of CSIM requires alterations to the C and C++ compilers respectively. Thus, CSIM simulation programs are not portable across compilers or machines.

Several research endeavors have produced packages that, together with a C++ compiler, facilitate the process-oriented view in the same manner as SIMULA. In [19], the task library of the C++ compiler is utilized to implement the simulation primitives and in [22], the lightweight threads package is utilized. Both of these approaches use packages that are not provided by all C++ compilers. Furthermore, the approach of using the task library was found to be less efficient than the approach taken by **SimPOL**[21].

5 Conclusions

We have presented **SimPOL**, our new system for process-oriented simulation. We detailed **SimPOL**'s language and illustrated its ease in simulating "real-world" applications using a factory floor simulation. Our presentation of the **SimPOL** translator outlined the steps in translation from a **SimPOL** program to a C++ program. For experimentation, we built a prototype of the **SimPOL** translator, and showed the resulting C++ code for one of the components of the factory floor simulation. We have ported our system using the factory floor simulation to both a Sun Sparc-10 and an IBM 80486-SX, using AT&T, GNU C++ and Borland compilers running under Unix and MS-DOS. Although we described our **SimPOL** system for C++, the technique can be applied to other object-oriented languages such as Smalltalk[7].

Since we implement **SimPOL** as a preprocessor, it is truly portable across both compilers and machines. Once an installation acquires the **SimPOL** preprocessor, simulation modeling is made simpler because the programmer is using a language with which he or she is familiar. Furthermore, if the installation acquires a new compiler, an upgraded version of the existing compiler or a completely new machine, the previously coded simulation models will easily port to the upgraded system without any alterations to the model itself. This is especially important since both compilers and machines are frequently upgraded.

We are extending our prototype implementation to include a debugger, and a tool for graphical presentation and monitoring of process-oriented simulations. We are also extending **SimPOL** to include statistical facilities to model real situations.

Appendix A: SimPOL Code for Factory Simulation

```
class WorkPiece;
class Drill;
class Lathe;
class Inspector;
queue<WorkPiece*>* drillQ;
queue<WorkPiece*>* threadQ;
queue<WorkPiece*>* inspectionQ;
queue<Drill*>* drills;
queue<Lathe*>* lathes;
queue<Inspector*>* inspectors;
extern long totalTime;
extern long totalWorkpieces;

class Drill:public Sim_Process {
public:
    Drill(char *n) :Sim_Process(n) {DRILL_TIME = 108;}
    virtual void Script();
private:
    int DRILL_TIME;
    WorkPiece *w;
};

class WorkPiece:public Sim_Process {
public:
    WorkPiece(char * n) :Sim_Process(n) { startTime = Time(); }
    virtual void Script();
private:
    long startTime;
    class Drill *d;
    class Lathe *l;
    class Inspector *ins;
};

class Lathe:public Sim_Process {
public:
    Lathe(char *n) : Sim_Process(n) {numPiecesDone = 0; }
    LATHE_ADJUST_TIME = 30;
    LATHE_ADJUST_AFTER = 10;
    THREAD_TIME = 12;}
    virtual void Script();
private:
    int LATHE_ADJUST_TIME;
    int LATHE_ADJUST_AFTER;
    int THREAD_TIME;
    int numPiecesDone;
    WorkPiece *w;
};

class Inspector:public Sim_Process {
public:
    Inspector(char *n) :Sim_Process(n) {INSPECTION_ TIME = 30;}
    virtual void Script();
private:
    WorkPiece *w;
    int INSPECTION_TIME;
};
```

```

void Drill::Script() {
    while(1) {
        if (!drillQ -> isEmpty() )
            w = drillQ -> get();
        else {
            drills -> put(this);
            Passivate();
            w = drillQ -> get();
        }
        Hold(DRILL_TIME);
        w -> Activate();
    }
}

void WorkPiece::Script() {
    drillQ->put(this);
    if (!drills->isEmpty()) {
        d = drills->get();
        d->Activate();
        Passivate();
    }
    else Passivate(); // drilling done
    threadQ->put(this);
    if (!lathes->isEmpty()) {
        l = lathes->get();
        l->Activate();
        Passivate();
    }
    else Passivate(); // threading done
    inspectionQ->put(this);
    if (!inspectors->isEmpty()) {
        ins = inspectors->get();
        ins->Activate();
        Passivate();
    }
    else Passivate(); // inspection done
    totalTime = totalTime + Time() - startTime;
    totalWorkPieces++;
    Passivate();
}

void Lathe::Script() {
    while (1) {
        if (numPiecesDone >= LATHE_ADJUST_AFTER) {
            Hold(LATHE_ADJUST_TIME);
            numPiecesDone = 0;
        }
        if (!threadQ->isEmpty())
            w = threadQ->get();
        else {
            lathes->put(this);
            Passivate();
            w = threadQ->get();
        }
        numPiecesDone++;
        Hold(THREAD_TIME);
        w->Activate();
    }
}

void Inspector::Script() {

```

```

while (1) {
    if (!inspectionQ->isEmpty())
        w = inspectionQ->get();
    else {
        inspectors->put(this);
        Passivate();
        W = inspectionQ->get();
    }
    Hold(INSPECTION_TIME);
    W->Activate();
}
}

simQueue *drillQ, *threadQ, *inspectionQ;
simQueue *drills, *lathes, *inspectors;

main() {
    int numLathes, numDrills, numInsp;
    int workPiecesPerHour, totalSimulationTime;
    Lathe *l;
    Drill *d;
    Inspector *ins;
    WorkPiece *w;
    inputParameters()
    drillQ = new simQueue;
    threadQ = new simQueue;
    inspectionQ = new simQueue;
    drills = new simQueue;
    lathes = new simQueue;
    inspectors = new simQueue;

    for (int i = 1; i <= numDrills; i++) {
        d = new Drill(i);
        drills->put(d);
    }
    for (i = 1; i <= numLathes; i++) {
        l = new Lathe(i);
        lathes->put(l);
    }
    for (i = 1; i <= numInsp; i++) {
        ins = new Inspector(i);
        inspectors->put(ins);
    }
    i = 0;
    while (i <= totalSimulationTime) {
        i++;
        w = new WorkPiece(i);
        w -> Activate();
        Hold( holdTime );
    }
    Hold(100000);
    printTotals();
}

```

References

- [1] AT&T C++ language system library manual. *Saber Software, Inc*, 1989.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [3] J. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL. *Communications of the ACM*, 6(1):1–17, 1962.
- [4] O. Balci. The implementation of four conceptual frameworks for simulation modeling in high-level languages. *Proceedings of the 1988 Winter Simulation Conference*, pages 287–295, 1988.
- [5] R. F. Belanger. MODSIM II - a modular, object-oriented language. *Winter Simulation conference*, pages 118–122, 1990.
- [6] A. Bloss. A functional approach to simulation programming. *Proceedings of the 1990 Winter Simulation Conference*, pages 214–220, 1990.
- [7] A. Boldberg and D. Robson. *Smalltalk-80 the language*. Addison Wesley, 1989.
- [8] J. N. Buxton. Simulation programming languages. *Proceedings of the IFIP Working Conference on simulation programming languages*, 1987.
- [9] J. S. Carson. Modeling and simulation worldviews. *Winter Simulation conference*, pages 18–23, 1993.
- [10] B. Diamond. Extend: A library-based, hierarchical, multi-domain modeling system. *Winter Simulation conference*, pages 240–248, 1993.
- [11] W. R. Franta. *The process view of simulation*. 1977.
- [12] B. Myhrhuag G. Birtwistle, O.-J. Dahl and K. Nygaard. *SIMULA Begin*. Studentlitteratur, Lund, 1980.
- [13] W. J. Garrison. A brief SIMSCRIPT II.5 tutorial. *Proceedings of Winter Simulation Conference*, pages 115–117, December 1990.
- [14] J. G. Goble. SIMOBJECT: From rapid prototyping to finished model – a breakthrough in graphical model building. *Winter Simulation conference*, pages 233–236, 1993.
- [15] C. R. Harrell. PROMOD (PROduction MODeler) for IBM PC's. *Supplementary Proceedings of the SCS Multiconference*, pages 65–70, 1989.
- [16] Borland International Inc. *Borland C++ User's Guide*. Scotts Valley, CA, 1992.
- [17] W. Kreutzer. *System simulation programming styles and languages*. Addison-Wesley, 1986.
- [18] G. Lamprecht. *Introduction to SIMULA 67*. Friedr, Vieweg & Sohn, Braunschweig/Wiesbaden, 1983.
- [19] B. A. Malloy and J. D. McGregor. A framework of classes for object oriented simulation. *Twenty Third Annual Modeling and Simulation Conference*, pages 1535–1541, April 1992.
- [20] B. A. Malloy and M.L. Soffa. Conversion of simulation processes to pascal constructs. *Software – Practice and Experience*, 20(2):pp 191–207, February 1990.
- [21] B. A. Malloy, J. M. Westall, and J. D. Dreuter. A simulation based study of the scalability of fine grained parallelism. *Technical Report #CU-94-27*, October 1994.
- [22] D. L. McCue and M. C. Little. Computing replica placement in distributed systems. *Second workshop on the management of replicated data*, pages 58–61, November 1992.

- [23] D. S. Mok and C. A. Funka-Lea. An interactive graphical modeling tool for performance and process simulation. *Winter Simulation conference*, pages 285–293, 1993.
- [24] P. Naur. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.
- [25] C.D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction to simulation using SIMAN*. McGraw-Hill, 1990.
- [26] A. B. Pritsker. *Introduction to Simulation and SLAM II*. Halsted Press, 1986.
- [27] D. P. Sanderson, R. Sharma, R. Rozin, and S. Treu. The hierarchical simulation language HSL: A versatile tool for process-oriented simulation. *acm Transactions on Modeling and Computer Simulation*, 1(2):113–153, April 1991.
- [28] T. J. Schriber. Perspectives on simulation using GPSS. *Proceedings of Winter Simulation Conference*, pages 5–13, December 1990.
- [29] H. D. Schwetman. CSIM:A C-based process-oriented simulation language. *Winter Simulation conference*, pages 387–396, 1986.
- [30] H. D. Schwetman. Using CSIM to model complex systems. *Winter Simulation conference*, pages 246–253, 1988.
- [31] H. D. Schwetman. CSIM reference manual. *Microelectronics and Computer Technology Corporation*, 1990.
- [32] H. D. Schwetman. Introduction to process-oriented simulation and CSIM. *Winter Simulation conference*, pages 154–157, 1990.
- [33] D. S. Smith and R. C. Crain. Industrial strength simulation using GPSS/H. *Proceedings of Winter Simulation Conference*, pages 165–171, December 1993.
- [34] R. M. Stallman. *Using and porting GNU CC (version 1.37.1)*. Free Software Foundation, Inc, Cambridge,MA, February 1990.
- [35] W. B. Thompson. A tutorial for modeling with the WITNESS visual interactive simulator. *Winter Simulation conference*, pages 228–232, 1993.
- [36] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, and C. H. Koster. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14(2):79–218, 1969.
- [37] N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 1971.
- [38] N. Wirth. Modula: A language for modular multi-programming. *Software-Practice and Experience*, 17:3–35, 1976.