Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

SIMULATION PROGRAM DEVELOPMENT BY STEPWISE REFINEMENT IN UNITY

Marc Abrams
Ernest H. Page
Richard E. Nance

Department of Computer Science
Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

ABSTRACT

Chandy and Misra's UNITY is a computational model and proof system suitable for development of parallel (and distributed) programs through step-wise refinement of specifications. UNITY supports the development of correct programs and the efficient implementation of those programs on parallel computer architectures. This paper assesses the potential of UNITY for simulation model specification and implementation by developing a UNITY specification of the machine interference problem with a patrolling repairman service discipline. The conclusions reached are that the UNITY proof system can assist formal verification of simulation models and the UNITY mappings of programs to various computer architectures offer some potential for assisting the automatic implementation of simulation models on parallel architectures. The paper gives some insights into the relationship of time flow mechanisms, parallel simulation protocols, and target parallel computer architectures.

1 INTRODUCTION

The automated support of simulation model development is entering the second decade as a topic of significant research interest. Approaches to computer assistance have sought a conceptual basis in artificial intelligence (Klahr 1985, Snyder and Macbulack 1988), general systems theory (Kim and Zeigler 1987, Murray and Sheppard 1987), software engineering (Henriksen 1983), and modeling methodologies (Balmer and Paul 1986, Nance 1981). In fact, the primary efforts in simulation support environments draw to varying degrees from all these conceptual sources. Balzer, Cheatham, and Green (1983, p. 41) describe the automation-based paradigm as separating implementation from specification so that maintenance is performed entirely on the latter. Automatic translation from a higher level specification to an efficient implementation is envisioned. This perspective on application development and support emphasizes the role of specification languages (see Stoegerer (1984) for an excellent survey) and the necessity for realizing an efficient implementation.

Simulation modeling represents a challenge for both model specification and implementation, and this work represents an effort to assess the potential of UNITY (Chandy and Misra 1988) for accomplishing both. (See the companion paper, Abrams, Page, and Nance (1991b), for a brief introduction to UNITY.) In addition, UNITY is intended for development of efficient parallel and distributed programs through step-wise refinement of specifications. This paper also assesses the potential of UNITY to derive efficient parallel simulation implementations.

A simulation program development methodology that uses UNITY is presented in Section 2 and applied to the Machine Repairman Problem in Section 3. Conclusions follow in Section 4.

2 UNITY-BASED METHODOLOGY

We propose that a simulation model be represented as a UNITY program by mapping simulation "attributes" and "events," as defined by Kiviat (Nance 1981), to UNITY "variables" and "assignment statements," respectively.

Assume that the "system and objectives definition" and "conceptual model" in Balci and Nance's simulation life cycle are completed (Balci 1989). We propose using a state transition diagram representation of the "communicative model" in the methodology to simplify the presentation. Starting at this point we propose the following methodology:

Step 1: This step specifies a simulation that captures the *order* of events that occur in the system, but ignores the absolute *time* at which events occur.

(For the machine repairman problem discussed in Section 3, this means capturing the correct state space and state transitions without regard to failure and repair rates or the rate at which the operator walks.)

Step 1A: Select a set of ***state variables***, enumerate all values of each state variable, and for each state variable enumerate all constraints on transitions that the system can make between the values of the state variable. Also specify which transitions are known to occur in a finite period of time.

(This paper uses one state transition diagram to represent each state variable, which permits a mechanical translation from the transition diagrams to the UNITY representation. However other representations, such as a single transition diagram, a Petri net, or English statements could be used.)

Verify that the states enumerated and the constraints on transitions match the conceptual model. Also verify that the list of constraints is complete (i.e., all invalid transitions are prohibited) and consistent (i.e., satisfying one constraint never leads to a violation of another constraint).

Step 1B: Express each output measure in terms of the holding time for a set of states. Verify that all output measures can be expressed in terms of the states selected in Step 1A.

Step 1C: Formalize the state transition diagram of Step 1A in UNITY. Verify that all transitions present (prohibited) in the diagram match transitions present (respectively, prohibited) in the UNITY specification.

Overall verification of Step 1: Verify that the communicative model and the UNITY specification agree in the following manner: State a set of properties that the communicative model implies, and use UNITY's proof system to show that the specification (i.e., the UNITY assertions of Step 1C and the additional properties of this step) implies these properties.

Step 2: Refine the simulation by mapping the order of events to a time scale. (In the machine repairman problem of Section 3, this means adding failure and repair rates and the rate at which the operator walks.) Verify that the refined specification meets the specification from Step 1.

Step 3: Derive a simulation program from the specification in Step 2. Formally verify using UNITY's proof system that the program meets the specification.

Step 4: Refine the simulation program by mapping the program to a particular (1) time flow mechanism, (2) sequential or parallel simulation protocol, and (3) sequential or parallel hardware architecture. We conjecture that these three must all be considered together to achieve an efficient program.

3 MACHINE REPAIRMAN PROBLEM

This paper applies the methodology of Section 2 to the classical machine interference problem (Cox and Smith 1961). In the problem, a set of N semiautomatic machines fail intermittently and are repaired by one or more technicians. Machine failure rates are assumed to follow a Poisson distribution with parameter λ. Upon arriving at a failed machine, a technician can repair the machine in a time period that is exponentially distributed with parameter μ. A variety of service disciplines are possible that specify how the technician selects a machine to repair.

The multiple repairman version of this problem should serve as an interesting benchmark for parallel simulation. The system being modeled contains concurrent behavior because machines fail independently, technicians after arriving at a machine repair machines independently. However the choice of service discipline introduces dependencies between the times that technicians arrive at machines that should frustrate efficient parallel execution of a simulation model.

This paper considers the patrolling repairman service discipline, in which a *single* technician services all machines (Nance 1971, p. 60). The technician traverses a path amongst the machines in a cyclic fashion $(1, 2, \ldots, N, 1, \ldots)$. The technician walks at a constant rate and only stops walking upon encountering a down machine. The technician takes constant time T to walk from one machine to the next. The model terminates when the number of machine repairs exceeds the constant *MaxRepairs*. This problem, hereafter referred to as the *machine repairman problem* (MRP), is chosen so that both the UNITY specification and program may be presented within the space available for this paper.

3.1 Illustration of Methodology Step 1

3.1.1 Step 1A: Select States and Specify Constraints on Transitions

Notation: Symbol N denotes the number of machines. Let m and n each denote an integer in the interval $[1, N]$ and represent machine numbers.

Machines: Each machine m is in one of two states: *up* or *down*. Associated with each m is a variable *m.state* that takes on values *up* or *down*. For convenience we employ variables $m.u$ and $m.d$, defined as:

$$\begin{aligned} m.u &\equiv (m.state{=}up) \\ m.d &\equiv (m.state{=}down) \end{aligned}$$

Therefore the value of *m.state* is *up* or *down* if m is up or down, respectively.

Technician: The technician is in one of $2N$ states: at machine 1, leaving machine 1, at machine 2, leaving machine 2, ..., at machine N, and leaving machine N.

To represent these $2N$ states, we associate with the technician a *single* state variable *loc* that takes on the $2N$ values $1, 1.5, 2, 2.5, \ldots, N, N+0.5$, respectively. For convenience we employ boolean variables $m.a$ and $m.l$, defined as:

$$\begin{aligned} m.a &\equiv (loc = m) \\ m.l &\equiv (loc = m + 0.5) \end{aligned}$$

Therefore the value of *loc* is 1 if the technician is at machine 1, the value is 1.5 if the technician is traveling from machine 1 to 2, the value is 2 if the technician is at machine 2, and so on.

Number of repairs: Symbol *NR* denotes the number of repairs to down machines that the technician has completed so far. Initially, $NR = 0$.

State Transition Diagrams: The state of the system is represented by $N+3$ state variables: $\forall m, m = 1, 2, \ldots, N$, *m.state*, *loc*, and *NR*. Constraints on the transitions between states that the system may make are represented using one state transition diagram for each state variable, as illustrated in Figures 1 through 3. Some transitions are labeled with Boolean functions of state variables not shown in the diagram, which means that the associated transition may only be taken if the Boolean function has value true. For example, in Figure 1 a machine can only go from a down state to an up state if the technician is present (e.g., the transition from state $m.d$ to $m.u$ occurs only if $m.a$ holds).

The diagram in Figure 1 specifies that an up machine may go down independently of the location of the technician or state of other machines, and a down machine may only go up when a technician is present. Figure 2 specifies that a technician that is at machine m advances to machine $m \oplus 1$ only if machine m is up.

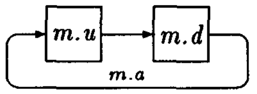

Figure 1: State Diagram Illustrating Variable *m.state*

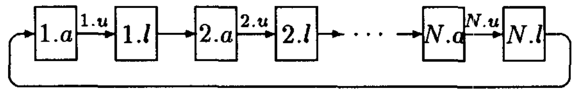

Figure 2: State Diagram Illustrating Variable *loc*

The diagram in Figure 3 uses double lines for transitions. We choose the convention that a double transition line can only occur simultaneously with the predicate labeling the transition becoming false. Therefore in Figure 3 the value of variable *NR* is incremented only when predicate $m.a \wedge m.d$ becomes false, corresponding to the technician leaving a machine and the machine going back up.

Modeling Simultaneity: Step 1A requires specifying states and legal and illegal transitions between states *without* specifying information about the duration of simulation time that can elapse between state transitions; the time may be zero or it may be positive. Constraints on times are added in Step 2.

Often a modeler knows that changes to multiple state variables must occur simultaneously; that is, zero simulation time must elapse between the setting of one variable and the setting of any other variable. This information may be incorporated into the specification either in Step 1A or in Step 2; either may be used as is convenient to the modeler:

1. in Step 1A specify *one* transition that changes the value of all state variables in the set, or

2. in Step 2 specify the holding time of instantaneous events to be zero.

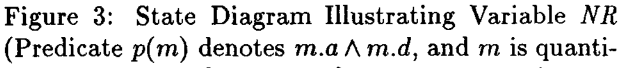

Figure 3: State Diagram Illustrating Variable *NR* (Predicate $p(m)$ denotes $m.a \wedge m.d$, and m is quantified over the set $\{0, 1, \ldots, N\}$ of all machines.)

Does it matter whether simultaneous state changes are specified in Step 1A or in Step 2? It is natural to assert that certain states have zero holding time in Step 2 and then implement a zero holding time in a simulation programming language; on the other hand specifying simultaneity in Step 1A permits formal verification of properties about simultaneous state changes using UNITY's proof system. (For example, one could verify that a machine never goes down while a technician is at the machine.) A definitive answer requires further investigation.

3.1.2 Step 1B: Express Output Measures Using Holding Times

Let us assume that the desired output measures are:

1. fraction of time which machine m is up,
2. fraction of time during which the technician is repairing machine m, and
3. fraction of time during which the technician is traveling.

In Step 1 we must show that the time intervals referred to in the output measures can be expressed in terms of the states identified in Step 1B. Calculation of measure 1 above is straightforward because state $m.u$ is the only state in which machine m is up. Calculation of measure 2 above is also straightforward because state $m.a$ is exactly the state in which the technician is repairing machine m. Calculation of measure 3 above is a little more complex. Define Boolean variable *traveling* as follows:

$$\textit{traveling} \equiv \langle \vee m :: m.l \rangle$$

Output measure 3 is simply the duration of simulation time for which predicate *traveling* has value *true*.

3.1.3 Step 1C: Formalize State Transition Diagrams in UNITY

Table 1 provides a set of rules that may be mechanically followed to generate a UNITY specification from each state transition diagram. The rules are applied to the MRP in Table 2.

Rule I formalizes a transition from a state S to a state S' without a Boolean function labeling the transition. In this case assertions (a)and optionally (b) must be added to the specification. Assertion (a) insures that when the state variable has value S, if it ever changes value, its next value must be S'. Optional assertion (b) is included if the following holds: when the state variable has value S, it must eventually change to value S'.

If diagram contains:	**Add to specification:**
Rule I: $\boxed{S}\rightarrow\boxed{S'}$	(a) S ***unless*** S' (b) $S \mapsto S'$ (optional)
Rule II: $\boxed{S}\xrightarrow{P}\boxed{S'}$	(c) $S \wedge \neg P$ ***unless*** $S \wedge P$ (d) S ***unless*** S' (e) $S \wedge P \mapsto S'$ (optional)
Rule III: $\boxed{S}\xRightarrow{\langle\exists m::P(m)\rangle}\boxed{S'}$	(f) $S \wedge \langle \forall m :: \neg P(m)\rangle$ ***unless*** $S \wedge \langle \exists m :: P(m)\rangle$ (g) $\langle \forall m :: S \wedge P(m)$ ***unless*** $S' \wedge \neg P(m)\rangle$ (h) $\langle \forall m :: S \wedge P(m) \mapsto S' \wedge \neg P(m)\rangle$ (optional)
Rule IV: Initial state is S	Initial condition $\Rightarrow S$

Table 1: Rules to Mechanically Map a State Diagram to a UNITY Specification (S and S' denote states, P and $p(m)$ denote Boolean valued predicates, $m \in W$, and W denotes any set. Rule III applies when at most one $p(m)$, for all $m \in W$, is true at any time.)

In Rule II, assertion (c) insures that when the state variable has value S and predicate P is false, then the state variable value remains constant as long as P remains false. Assertion (d) is similar to (a). Optional assertion (e) is included if the following holds: when the state variable has value S and P holds, then eventually the variable must change to value S'.

In Rule III, assertion (f) is similar to (c), but it is generalized to handle multiple predicates on the arc. Assertion (g) generalizes (d) to capture the essence of what makes the double arrow transition differ from the single arrow: when the state variable has value S and predicate $P(m)$ holds for some m, after the next state transition (in this or another state diagram), either the condition continues to hold or else the transition changes the state variable value to S' and $P(m)$ is now false. Optional assertion (h) is included if, when the state variable has value S and predicate $P(m)$ holds for some m, the transition described by (g) must eventually occur.

Rule IV simply adds the initial condition of each

Specification includes:	**Due to state diagram fragment:**	**From Figure:**
MRP1: $m.u$ *until* $m.d$	$\boxed{m.u} \rightarrow \boxed{m.d}$	1
MRP2: $m.d$ *unless* $m.u$ MRP3: $m.d \wedge \neg m.a$ *unless* $m.d \wedge m.a$ MRP4: $m.d \wedge m.a \mapsto m.u$	$\boxed{m.d} \xrightarrow{m.a} \boxed{m.u}$	1
MRP5: $m.a$ *unless* $m.l$ MRP6: $m.a \wedge m.d$ *unless* $m.a \wedge m.u$ MRP7: $m.a \wedge m.u \mapsto m.l$	$\boxed{m.a} \xrightarrow{m.u} \boxed{m.l}$	2
MRP8: $m.l$ *until* $(m \oplus 1).a$	$\boxed{m.l} \rightarrow \boxed{(m \oplus 1).a}$	2
MRP9: $NR = k \wedge \langle \forall m :: \neg(m.a \wedge m.d) \rangle$ *unless* $NR = k \wedge \langle \exists m :: m.a \wedge m.d \rangle$ MRP10: $NR = k \wedge m.a \wedge m.d$ *until* $NR = k + 1 \wedge \neg(m.a \wedge m.d))$	$\boxed{k} \xRightarrow{\langle \exists m :: m.a \wedge m.d \rangle} \boxed{k+1}$	3
MRP11: Initial condition $\Rightarrow NR = 0$		3

Table 2: Complete Unity Specification for MRP

state diagram, if any, to the specification.

The set of rules in Table 1 are insufficient to formalize any state transition diagram. In particular, Rules I and II must be modified to handle the case of two or more output arcs from a value S. Rule III must be modified to handle the case of two or more output arcs to *different* values; in its present form Rule III handles multiple arcs to the *same value* S'. These generalizations are unnecessary for formalization of the machine repairman problem and are omitted.

Applying the rule of Table 1 to the transition diagrams of Figures 1 to 3 yields the UNITY specification of the MRP shown in Table 2. For example, MRP1 states that a machine that is up must eventually go directly to a down state.

Note that the definition of logical relation *until* has been used in Rules I and III whenever the optional assertion is included to reduce the number of assertions that comprise the specification. In all UNITY formulas in the paper, universal quantification over the values of variable *loc* is assumed, unless the quantification is explicit. Hence all formulas except MRP11 in Table 2 hold for $m = 1, 2, \ldots$.

The only verification necessary for Step 1C is to insure that the rules from Table 1 have been correctly applied.

3.1.4 Overall verification of Step 1

The specification of Table 2 is verified by stating additional properties and using UNITY's proof system to formally show that the specification implies these properties. Inability to prove the properties implies that the specification is incomplete or incorrect, or that the properties themselves do not hold for the system. Carrying out such a proof does not guarantee the correctness of the specification, but does increase our confidence in the specification. In fact, in writing this paper our original statement of the specification omitted several properties shown in Table 2.

P1 :	$m.d \mapsto m.u$
P2 :	$m.a$ *until* $m.l$
P3 :	*true* $\mapsto NR \geq MaxRepairs$

Figure 4: Properties of MRP Used to Formally Verify Specification Correctness

We give three properties (Figure 4) which are proved in Abrams, Page, and Nance (1991a). First, when a machine goes down, it is eventually repaired and comes back up (P1). Second, when the technician is at a particular machine, he remains at that machine until, eventually, he leaves that machine (P2). Third, the value of variable *NR* eventually exceeds any constant *MaxRepairs* (P3).

3.2 Illustration of Methodology Step 2

In Step 2, the specification of Step 1 is augmented by two additional assertions on the holding time of certain states, specified in units of simulation time. Before stating the assertions, two additional variables are necessary.

A *sequence* is a data type commonly employed in UNITY specifications, and represents a list of items with a first element and a last element. If s denotes a sequence, then Head(s) is the first element of the sequence, and Tail(s) is the sequence obtained by deleting Head(s).

The specification represents "calls to a random number generator" by referring to a sequence whose elements are a list of random variates returned by the random number generator. Let sequence $m.\lambda$ denote a list of random variates representing the sequence of times for which machine m remains up. Let sequence $m.\mu$ denote a list of random variates representing the sequence of repair times of machine m.

The additional assertions are:

1. State $m.u$ has holding time Head(m.λ).
2. State $m.d \wedge m.a$ has holding time Head(m.μ).
3. State $m.u \wedge m.a$ has holding time zero.

UNITY has no notion of "time"; therefore these assertions cannot be formalized in UNITY.

Verification: The specification of Step 1 is subsumed by the specification of Step 2.

3.3 Illustration of Methodology Step 3

The specification of Table 2 is implemented by program MRP, shown in Figure 5.

Verification: Formal proof that the code meets the specification in Table 2 can be carried out, but is not presented in this paper. Proof that the output measures are correctly computed requires formulating and proving a suitably strong invariant.

It is impossible to prove the time-in-state assertions from Section 3.3 using the current proof system of UNITY. UNITY's computational model of fairly interleaved, atomic execution of statements permits no notion of simultaneity, which means that fundamental changes to UNITY are required to carry out these proofs.

3.4 Illustration of Methodology Step 4

In this section we explore how different time flow algorithms may be added to a UNITY simulation specification of the form given in Step 3. In particular, we consider two classical time flow mechanisms: fixed time increment and Time-of-Next-Event.

UNITY advocates program development by stepwise refinement of specifications, with the transformation from the most refined specification to a program written in a programming language being the "most mechanical and least creative part of the process" (Chandy and Misra 1988, p. ix). To apply this philosophy to the simulation program development cycle, we must have a way to refine the specification of Step 3 into the specification of Step 4 by adding a time flow mechanism. Step 4 is necessary *only* as we move toward implementation and is not necessary for specification of model behavior in its most basic sense (i.e. *what* the model does rather than *how* the model accomplishes what it does). Therefore the addition of a time flow mechanism in Step 4 should be accomplished with minimal (ideally no) perturbations of the Step 3 specification. We demonstrate below that this can be accomplished using the UNITY concept of *superposition*.

3.4.1 Superposing Fixed Time Increment

First we consider the specification of the fixed time increment time flow mechanism.

Symbol Δ denotes an integer value of simulation time, representing a time increment; the value of Δ is fixed during simulation. Recall from Figure 5 that *SysTime* is a program variable containing the current simulation time. The fixed time increment algorithm consists of two phases:

1. Execute any statements (events) whose alarms have gone off at the current value of *SysTime*.
2. Set *SysTime* to *SysTime* + Δ.

In order to add the above two phase algorithm to the Step 3 specification (Figure 5) we must devise a means to insure that *all* statements whose alarms have gone off at the current value of *SysTime* are executed *before SysTime* is incremented. (Because UNITY does not specify sequencing of statements, we must add something to enforce the two phase sequencing.)

Let the UNITY program of Step 3 contain S statements in the **assign** section ($S = 5$ in Figure 5). To enforce the two phase algorithm, we first number the statements in the program of Step 3 by the integers 1,2,...,S. Next we add array $A[1..S]$. Initially, all elements of array A are zero. Each statement s_i numbered i (for $1 \leq i \leq S$) is transformed to $s_i \;||\; A[i] := 1$. When all elements of array A are one, *SysTime* can be incremented. When system time is incremented (in the superposed program) all elements of array A are set to zero. (Note that an assignment statement of the form $x := e$ if b in Figure 5 is a shorthand for $x := e$ if $b || x := x$ if $\neg b$. Therefore the statement is executed even though b is false.)

The superposed program is formalized in Figure 6. Note that Figure 6 works with *any* simulation specification that results from Step 3.

```
program MRP {simulate the MRP}
declare
  constants N=...; MaxRepairs=...; T=...;
  types alarm = integer;
  variables
    m           : integer              {machine number; integer in [1,N]}
    State[N]    : (up, down)           {enumerated type}
    Loc         : (1,1.5,...,N,N+0.5)  {enumerated type}
    NR          : integer              {number of completed repairs}
    SysTime     : integer              {current simulation time; read only }
    Failure[N]  : alarm                {Failure[m] = time machine m next fails if Failure[m]>SysTime}
    Arrival[N]  : alarm                {Arrival[m] = time technician next arrives at machine m, if m⊖0.5
                                       = Loc, otherwise time when technician arrived at machine m}
    Finish[N]   : alarm                {Finish[m] = time machine m goes back up if State[m]=down and
                                       Loc=m, otherwise time of last repair completion}
    λ[N]        : sequence of integer  { sequence of random variates representing time between failures}
    μ[N]        : sequence of integer  {sequence of random variates representing repair times}

always
  term = NR ≥ MaxRepairs

initially
    SysTime = 0.0 || NR = 0 || Loc=1.5 {technician initially leaving machine 1}
  || ⟨ || m : 1 ≤ m ≤ N :: State[m] = up ⟩ {initially all machines are up }
  || ⟨ || m : 1 ≤ m ≤ N :: Failure[m] = SysTime + Head(λ[m]) || λ[m] = Tail(λ[m]) ⟩
  || ⟨ || m : 1 ≤ m ≤ N :: Arrival[m] = SysTime + T if m = Loc ⊕ 0.5  ~  -∞ if m ≠ Loc ⊕ 0.5 ⟩

assign
  {Arrival: Update location; schedule finish if machine is down, else schedule arrival at next machine. }
  □ ⟨ || m : 1 ≤ m ≤ N :: Loc := m if SysTime = Arrival[m] ∧ ¬term
      □ Finish[m], μ[m] := SysTime+Head(μ[m]), tail(μ[m]) if Loc=m ∧ State[m]=down ∧ ¬term
      □ Loc, Arrival[(m ⊕ 1)] := m ⊕ 0.5, SysTime+T if Loc=m ∧ State[m]=up ∧ ¬term
    ⟩

  {Finish: Increment NR, set machine state to up, schedule next failure, update technician's location
  and schedule arrival at next machine}
  □ ⟨ || m : 1 ≤ m ≤ N :: NR, Arrive[m ⊕1], Failure[m], State[m], Loc, λ[m] :=
      NR+1, SysTime+T, SysTime+Head(λ[m]), up, m ⊕0.5, Tail(λ[m]) if SysTime=Finish[m] ∧ ¬term
    ⟩

  {Failure: Set machine's state to down. }
  □ ⟨ || m : 1 ≤ m ≤ N :: State[m] := down if SysTime=Failure[m] ∧ ¬term ⟩

end { MRP }
```

Figure 5: UNITY Code for MRP

```
Program FTI_TFM
declare A[S] : integer
initially ⟨i : 1 ≤ i ≤ A[i] = 0⟩
transform
    each statement s in the underlying program
    to s || A[i] := 1 where i is the lexical state-
    ment number of s.
add to always section
    update = ⟨∧i : 1 ≤ i ≤ S :: A[i] = 1⟩
add to assign section
    ⟨|| i :: 1 ≤ i ≤ S :: A[i] := 0    if update ⟩
    || SysTime := SysTime + Δ    if update
end { FTI_TFM }
```

Figure 6: Specification of Fixed Time Increment Time Flow Mechanism

This superposition can be accomplished with no changes to the underlying specification (other than the ones addressed by the superposition program of course). So, for the fixed time increment time flow mechanism we seem to have achieved our ideal.

3.4.2 Superposing Time-of-Next-Event

Next we sketch a method to add the next event time flow mechanism to a Step 3 program. As in the fixed time increment method, we assign each statement in the assign section an integer identification number. These numbers serve as event numbers. We add an EventList and a variable called CurrentEvent. Recall that m is an integer in $[1, \ldots, N]$ denoting a machine number. EventList is a list of triples (time, event number, m). The statements which set alarms in Figure 5 now append triples to EventList. The time flow mechanism superimposed on the program sets *SysTime* to the time component of a triple of EventList that is less than or equal to the time component of all other triples. This triple's event number is stored in CurrentEvent. Finally, We add to each statement s_i in the assign section the condition "if CurrentEvent=s_i."

This superposition fails to achieve our goal of not modifying the specification in Step 3 in order to add a time flow mechanism. Therefore the Step 3 specification is biased towards Fixed Time Increment. One way to rectify this is to modify the definition of superposition in UNITY, which would require the proof system to be extended. A second way to rectify this would be to choose a representation in Step 3 not based on alarms that maps as easily to Fixed Time Increment and to Time-of-Next-Event.

3.4.3 Mapping Specification to a Protocol and Architecture

Mapping a simulation specification to a simulation protocol is an open problem. Mapping of UNITY specifications to architectures is discussed by Chandy and Misra (1988, Chapter 4), and applies to simulation specifications.

We propose that jointly mapping a simulation specification to a time flow mechanism, sequential or parallel simulation protocol, and sequential or parallel hardware architecture may be necessary to achieve an efficient program. In terms of UNITY, the result of all three mappings is a set of constraints on when assignment statements (corresponding to simulation events) can be executed.

The simplest joint mapping maps a simulation specification to a fixed time increment time flow mechanism, a synchronous parallel simulation protocol, and a synchronous shared-memory computer architecture. All three mappings produce the same constraint: that all events (assignment statements) are executed each time the clock is incremented.

However, mappings to other time flow mechanisms, parallel simulation protocols, and architectures are more complex and constitute an open problem.

4 CONCLUSIONS

Step 1 of the proposed methodology dictates that the *order* of events in a conceptual model be correctly specified without regard for the particular times at which events occur. The justification is that one often wishes to "get the simulation logic correct." Based on the example in Section 3.1, UNITY works well for this job.

Step 2 (mapping the order of events to a time scale) requires a modification of UNITY to add notation for the holding time of certain states. We introduced such a notation in Section 3.2. However, in order to prove any properties about timings, the UNITY proof system must be extended, which is likely to be a difficult task.

Step 3 (deriving a simulation program from a specification) in Section 3.3 is straightforward. Again, we cannot formally verify the correctness of the timing properties without an extension of the proof system to handle time.

Step 4 (mapping the program to a particular time flow mechanism, sequential or parallel simulation protocol, and sequential or parallel target architecture) requires additional research to accomplish. Based on the example in Section 3.4 of mapping the MRP to fixed time increment as well as the Time-of-Next-Event mechanisms, we believe UNITY is sufficient to handle Step 4.

Based on the specification example in this paper, UNITY could help simulation modelers in three areas:

Model verification: UNITY provides a comprehensive proof system of both safety and progress properties, which can be applied to verifying properties of simulation models. Our experience in proving the properties of Figure 4 is that UNITY proofs are fairly mechanical, but can be time consuming. Following are some specific examples of where the proofs are time consuming.

(a) Applying induction: A key to the proof that down machines are eventually repaired (P1) is establishing by an induction proof that after a machine goes down, the technician keeps getting "closer" to the failed machine, until eventually he is at the failed machine. Induction is required whenever we want to draw a conclusion about a *sequence* of state transitions, given a specification describing only *single step transitions*, as Table 2 does. Figuring out how to fit the induction theorem to this intuition did require some time on the part of the authors.

(b) Constructing chain of deductions: In general the authors spent much of their time playing with the more than thirty theorems in the UNITY book (Chandy and Misra, Chapter 3) to construct the formal chain of deductions required for each proof. This process is somewhat analogous to what an undergraduate student does in a calculus class, as he browses through a table of integrals and a list of trigonometric identities in trying to symbolically integrate a function. However a theorem proving system might alleviate this problem.

(c) Devising invariants: This paper does not present a proof that the simulation code (Figure 5) meets the specification. However, proofs of code generally require invariants to be formulated, which takes some creativity. This is analogous to integrating a function by guessing the antiderivative.

As our experience with UNITY grows, we expect the time required for items (a) and (b) listed above to decrease.

Automation-based paradigm: The fact that we could give, in Table 1, a set of rules to map certain state transition diagrams to a UNITY specification in a mechanical manner is encouraging. We believe that additional rules can be developed to represent any state transition diagram, as well as other forms of model representation (e.g., Petri nets). If UNITY grows in popularity, a rich set of methods to map UNITY programs to target architectures may be developed. By identifying the correspondence between simulation modeling and UNITY programs, a model development environment using the automation-based paradigm could apply the UNITY architecture mappings for simulation models to assist in construction of parallel simulation programs.

Mapping specification to time-flow mechanism, parallel protocol, and target machine architecture: An important lesson from the exercise in this paper is that mapping a simulation specification to a time-flow mechanism, a parallel simulation protocol (e.g., conservative-synchronous, conservative-asynchronous, optimistic), and a target machine architecture are intimately connected. All three correspond to specifying constraints on *when* to execute statements in a UNITY program. Perhaps all three must be done jointly during the program development cycle to obtain a sufficiently efficient program.

Efficient parallel execution of a simulation model implies consideration of the constraints imposed by each combination of computer architecture, time flow mechanism, and parallel simulation protocol, which leads to an enormous design space. An additional complication is that many of these constraints are input data dependent; thus a correct temporal ordering of events cannot be predicted before execution. This exposes one reason why parallel discrete-event simulation programming is a fundamentally hard problem.

REFERENCES

Abrams, M., E. H. Page, and R. E. Nance. 1991a. Linking Simulation Model Specification and Parallel Execution Through UNITY. Technical Report TR 91-14, Computer Science Department, VPI&SU, May 1991.

Abrams, M., E. H. Page, and R. E. Nance. 1991b. Linking Simulation Model Specification and Parallel Execution Through UNITY. *Proceedings of the 1991 Winter Simulation Conference.*

Balci, O. 1989. How to Assess the Acceptability and Credibility of Simulation Results. *Proceedings of the 1989 Winter Simulation Conference,* eds. E. A. MacNair, K. J. Musselman, and P. Heidelberger, 62-71.

Balmer, D.W. and R.J. Paul. 1986. CASM - The Right Environment for Simulation. *Journal of the Operational Research Society 37*, 443-452.

Balzer, R. T. E. Cheatham, Jr., and C. Green. 1983. Software Technology in the 1990s: Using a New Paradigm. *Computer* **16** (11), November, 39-45.

Chandy K. M. and J. Misra. 1988. *Parallel Program Design: A Foundation.* Reading, MA: Addison Wesley.

Cox D. R. and W. L. Smith. 1961. *Queues*, Methuen and Company, Ltd.

Henriksen, J. O. 1983. The Integrated Simulation Environment (Simulation Software of the 1990s). *Operations Research* **31** (6), November-December, 1053-1073.

Kim, T.G. and B.P. Zeigler. 1987. The DEVS Foundation: Hierarchical, Modular Systems Specifications in a Object Oriented Framework. *Proceedings of the 1987 Winter Simulation Conference*, eds. A. Thesen, H. Grant, and W.D. Kelton, 559-566.

Klahr, P. 1985. Expressibility in ROSS: An Object-Oriented Simulation System. *AI Applied to Simulation: Proceedings of the European Conference at the University of Ghent*, 136-139.

Murray, K.J. and S.V. Sheppard. 1987. Automatic Model . Synthesis: Using Automatic Programming and Expert Systems Techniques Toward Simulation Modeling. *Proceedings of the 1987 Winter Simulation Conference*, eds. A. Thesen, H. Grant, and W.D. Kelton, 534-543.

Nance, R. E. 1971. On Time Flow Mechanisms for Discrete System Simulation. *Management Science* **18**, (1), September, 59-73.

Nance, R. E. 1981. The Time and State Relationships in Simulation Modeling. *Communications of the ACM* **24**, (4), 173-179.

Snyder, J. and G.T. Macbulack. 1988. Intelligent Simulation Environments: Identification of the Basics. *Proceedings of the 1988 Winter Simulation Conference*, eds. M. Abrams, P. Haigh, and J. Comfort, 359-363.

Stoegerer, J.K. 1984. A Comprehensive Approach to Specification Languages, *The Australian Computer Journal* **16**(1), February, 1-13.

AUTHOR BIOGRAPHIES

MARC ABRAMS is an assistant professor of Computer Science Department at Virginia Polytechnic Institute and State University. His research interests include parallel simulation, software performance analysis, and communication protocols. He received his Ph.D. from the University of Maryland in 1986. He serves as Program Chair for the 1992 SCS Parallel and Distributed Simulation (PADS) workshop.

ERNEST H. PAGE is a Ph.D. student in the Department of Computer Science at Virginia Polytechnic Institute and State University (VPI&SU). He received B.S. and M.S. degrees in Computer Science from VPI&SU in 1988 and 1990. He has served as the Chairman of the ACM Student Chapter at VPI&SU during the 1988-89 academic year. His research interests include simulation model development environments, parallel and distributed simulation, and software engineering. He is a member of ACM, ACM SIGSIM, IEEE CS, SCS, and Upsilon Pi Epsilon.

RICHARD E. NANCE is the RADM John Adolphus Dahlgren Professor of Computer Science and the Director of the Systems Research Center at Virginia Polytechnic Institute and State University (VPI&SU). He received B.S. and M.S. degrees from N.C. State University in 1962 and 1966, and the Ph.D. degree from Purdue University in 1968. He as served on the faculties of Southern Methodist University and VPI&SU, where he was Department Head of Computer Science, 1973-1979. Dr. Nance has held research appointments at the Naval Surface Weapons Center and at the Imperial College of Science and Technology (UK). The author of over 80 papers on discrete event simulation, performance modeling and evaluation, computer networks, and software engineering, Dr. Nance is the founding Editor-in-Chief of the *ACM Transactions on Modeling and Computer Simulation* and served as Program Chair for the 1990 Winter Simulation Conference. He is a member of Sigma Xi, Alpha Pi Mu, Upsilon Pi Epsilon, ACM, IIE, ORSA, and TIMS.