Simulation Model Development and Analysis in UNITY

Ernest H. Page
The MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102

Marc Abrams
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Abstract

We evaluate UNITY – a computational model, specification language and proof system defined by Chandy and Misra [5] for the development of parallel and distributed programs – as a platform for simulation model specification and analysis. We describe a UNITY-based methodology for the construction, analysis and execution of simulation models. The methodology starts with a simulation model specification in the form of a set of *coupled state transition systems*. Mechanical methods for mapping the transition systems first into a set of formal assertions, permitting formal verification of the transition systems, and second into an executable program are described. The methodology provides a means to independently verify the correctness of the transition systems: one can specify properties formally that the model should obey and prove them as theorems using the formal specification. The methodology is illustrated through generation of a simulation program solving the machine interference problem using the Time Warp protocol on a distributed memory parallel architecture.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development – *modeling methodologies*;; I.6.8 [**Simulation and Modeling**]: Types of Simulation – *parallel, distributed*

General Terms: simulation specification, simulation verification, parallel simulation protocols, UNITY

Additional Key Words and Phrases:

1 Introduction

Model specification is often viewed as a critical process within the simulation model life cycle [2]. The specification process is typically realized using a *specification language*. Dozens of specification languages have been proposed for general software system development, and numerous simulation-specific specification languages have also appeared, e.g. [19]. Many of these software and simulation specification languages are formal in nature. Precision in language syntax and semantics facilitates diagnostic analysis that can assist in the determination of model correctness [17, 19].

Although methods for formally reasoning about sequential, general purpose computer programs were first proposed twenty-five years ago [10], few methods permit formal reasoning about simulation model specifications. Reasoning about simulations requires methods that accommodate: (1) simulation time, (2) prioritization of events or actions scheduled for the same simulation time, and (3) output measures. While problem (2) and problems similar to (1) are addressed in the areas of real-time systems [1, 23] and communication protocols [25, Section 6.1], no existing methods address problem (3). Zeigler's DEVS formalism [29, 30, 31] provides some mechanisms for specifying and analyzing properties of simulation output measures, but not in the form of a general-purpose formal reasoning system and automated theorem proving mechanism.

This paper suggests an approach for simulation model development that admits the application (and extension) of a general-purpose formal reasoning system. The suggested methodology pursues automated construction and verification of sequential and parallel simulation programs from a communicative model using the UNITY computation model and proof system [5] which is reviewed in Section 2. The methodology supports multiple time flow mechanisms, execution protocols, and target computer architectures. The approach is described in six steps. The first step generates a communicative model in the form of a coupled state transition system (CSTS), which is formally defined in Section 3. The CSTS is an algebraic specification, defining all possible transitions among simulation model state variable values, and all constraints

that dictate when each transition may occur. The second step is the automated generation of a UNITY program from the CSTS. The third step provides automated generation of UNITY assertions from the CSTS (Section 6). In the fourth step, a time flow mechanism is superposed on the UNITY program (Section 7). A novel aspect of the methodology is that a simulation modeler can state (some) properties that a correct communicative model must possess, and then use the UNITY proof system to formally verify the correctness of the CSTS. Use of the rules guarantees that the program meets the specification embodied by the CSTS. The fifth step maps the program resulting from step four to a target computer architecture. One heuristic for this step is given in Section 8. The final step in the methodology maps the program resulting from step five to a simulation protocol for sequential or parallel execution. A mapping to the Time Warp protocol is outlined in Section 9. The methodology is summarized in Section 10, and illustrated with the machine interference problem.

As a work in formal methods, the paper necessarily contains a significant degree of notation. We have made every effort to give a "plain English" description of the key notation and formulas in the paper and we have included a reference for the notation in Appendix 11. However, some level of comfort with formalism on the part of the reader is probably required. We also concede that, as is often the case in developing formal methods, the initial scaleability of the methods suggested here is somewhat limited, and certainly this factor hampers the immediate practicality of the approach. Our objective in this work is to pursue and foster the development of highly automated systems that enable users to be insulated from many of the underlying complexities of reasoning about simulation models. We consider the methods proposed here an early step toward that goal.

2 Introduction to UNITY

Several factors mitigate in favor of Chandy and Misra's UNITY as a platform for formal simulation model development and analysis:

- UNITY permits program development through stepwise refinement, desirable for simulation program development.
- The UNITY computation model is based on a state transition system, which underlies other proof systems (e.g., [22, 25]). Transition systems do not explicitly specify control flow (e.g., *while* and *if* statements in imperative programming languages), which is desirable for two reasons. First, different parallel computers use different forms of control flow. Second, sequential programmers are accustomed to over-specifying control flow. Efforts to automatically transform sequential FORTRAN programs to parallel programs, for example, typically require code analysis to identify the control flow constraints that can be relaxed.
- UNITY proof rules are based on temporal logic, and real time extensions to temporal logic have been proposed (e.g., [13]) that could be used to reason about simulation time and ordering of events and actions scheduled for the same simulation time.
- UNITY assertions permit algebraic, rather than operational, specification. Algebraic specification naturally captures what a model is to do without specifying how it is to be done, and hence is well suited to model representation.
- A proof system suitable for mechanical verification of UNITY proofs exists [12].

UNITY provides a means to systematically develop and prove properties about programs for a wide variety of applications and computer architectures. Architectures considered include sequential processors, synchronous and asynchronous shared-memory multiprocessors, and message-based distributed processors.

UNITY supports program development by stepwise refinement of specifications. The final specification is implemented as a program, and the program may be refined further if necessary. During early stages of refinement, correctness is a primary concern. Considerations for efficient implementation on a particular architecture are postponed until later stages of refinement. Thus, one may specify a program that may ultimately be implemented on many different architectures. This process can be envisioned as generating

a tree of specifications, in which the root is a correct and, ideally, entirely architecture-independent specification, and each leaf corresponds to a correct specification of an efficient solution for a particular target architecture. Development of a correct UNITY program requires, at each stage of refinement, proof that the refined specification implies the previous specification. In addition, one must prove that the program derived from the most refined specification meets that specification.

2.1 Computational Model

"A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple assignment statements"[5, p. 9]. The UNITY computational model at first appears to be somewhat unconventional. (The *state* of a program after some step of the computation is the value of all program variables):

> A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following *fairness* rule: Every statement is selected infinitely often [5, p. 9].

"Infinitely often" means that at any point during program execution, every statement in the program must be executed at some point in the future. (Note that the computational model represents asynchronous execution of assignments in a parallel computer by interleaved execution.)

A UNITY program never terminates. However, a program may reach fixed point (FP), which is a computation state in which execution of any assignment statement does not change the state. At FP, the left and right hand side of each assignment statement are equal, and an implementation can thereafter terminate the program.

The UNITY computational model appears conventional if viewed as a set of state transition machines, where execution of an assignment statement corresponds to a transition.

The UNITY goal of postponing questions of efficiency and architecture to late in the refinement process is achieved by saying very little about the *order* in which assignments are executed during early specification stages, and by including control flow in the form of a detailed execution schedule of assignment statements efficient for a particular target architecture as a last step in program development.

2.2 Programming Logic

UNITY contains a formal specification technique that uses certain notation and logical relations. Let p and q denote arbitrary predicates, or Boolean valued functions of the values of program variables. Let s denote an assignment statement in a program. The assertion $p \Rightarrow q$ is read "if p holds then q holds." The assertion $\{p\}s\{q\}$ denotes that execution of statement s in any state that satisfies predicate p results in a state that satisfies predicate q, if execution of s terminates.

The notation $\langle op\ var\text{–}list\ :\ boolean\text{–}expr\ ::\ assertion\ \rangle$ denotes an expression whose value is the result of applying operator *op* (e.g., quantifiers $\forall$ (for all) and $\exists$ (there exists), $+$ (sum), max, logical operators $\wedge$ (and) and $\vee$ (or)) to the set of expressions obtained by substituting all instances of variables in the *var–list* that satisfy the *boolean–expr* in the *assertion.* For example, if i denotes an integer, $\langle +i : 1 \leq i \leq N :: i \rangle$ is an expression whose value is $\sum_{i=1}^{N} i$.

UNITY defines three fundamental logical relations: *unless*, *ensures*, and *leads-to*. The definitions below are those of Chandy and Misra.[5, Ch. 3]

Unless: The assertion "$p\ unless\ q$" means that if p is true at some point in the computation and q is not, in the next step (i.e., after execution of a statement) either p remains *true* or q becomes *true*. Therefore either q never holds and p continues to hold forever, or q holds eventually (it may hold initially when p holds) and p continues to hold at least until q holds. Formally, $p\ unless\ q \equiv \langle \forall s\ :\ s\ in\ F\ ::\ \{p \wedge \neg q\}\ s\ \{p \vee q\} \rangle$, where s is quantified over all statements in a given program.

program *sort*
assign
$\langle \Box\, i \;:\; 0 \leq i < N \;::\; A[i], A[i+1] \;:=\; A[i+1], A[i]$ if A[i]>A[i+1]$\rangle$
end $\{sort\}$

Figure 1: Sort Array A into Ascending Order.

program <*name*>
  declare <*var–decl–list*>
  initially <*initial–list*>
  always <*initial–list*>
  assign <*stmt–list*>
end { <*name*> }

Figure 2: UNITY Program

Ensures: The assertion "p *ensures* q" means that if p is *true* at some point in the computation, p remains *true* as long as q is false, and eventually q becomes true. This implies that the program contains a single statement whose execution in a state satisfying $p \wedge \neg q$ establishes q. Formally, $p\, ensures\, q \;\equiv\; p\, unless\, q \wedge \langle \exists s \;:\; s\; in\; F \;::\; \{p \wedge \neg q\}\; s\; \{q\}\rangle$where s is quantified over all statements in a given program.

Leads-to: Leads-to is denoted by the symbol $\mapsto$. The assertion "$p \mapsto q$" means that if p becomes *true* at some point in the computation, q is or will be *true*. The formal definition of leads-to is somewhat lengthy, and is not given here.

Based on the three fundamental logical relations *unless, ensures,* and *leads-to,* additional relations may be defined. We discuss two additional relations: *until* and *invariant*.

Until: The assertion "$p\; until\; q$" means that p holds at least as long as q does not and that eventually q holds. The assertion $p\; until\; q$ relaxes the requirement that execution of exactly *one* statement in a state satisfying $p \wedge \neg q$ establishes q. Formally, $p\; until\; q \;\equiv\; (p\; unless\; q) \wedge (p \mapsto q)$.

Invariant: An invariant property is always *true*: All states of the program that arise during any execution sequence of the program satisfy all invariants. Formally, q is invariant $\equiv$ (initial condition $\Rightarrow$ $q) \wedge q\; unless\; false$.

2.3 Program Notation

UNITY generates two artifacts during the specification process: a list of assertions using the logical relations introduced in Section 2.2 and an implementation of the assertions in a UNITY program. The program syntax is shown in Figure 2.

The **declare** section specifies the variables used in the program and their types. The **initially** section specifies the initial value of program variables. The **always** section can be thought of as defining functions; function names appear on the left hand side of the symbol "=". The **assign** section contains assignment statements performed during program execution.

A <*var–decl–list*> is a list of variable declarations expressed using the syntax of the programming language Pascal. The <*initial–list*> and <*stmt–list*> are identical in syntax, except that "=" and ":=" are used, respectively. A <*stmt–list*> has the form <*stmt*> $\Box$ <*stmt*> $\Box \cdots \Box$<*stmt*>. The symbol "$\Box$" separates statements. A <*stmt*> is either a quantified statement list or a single statement. A quantified statement list, $\langle \Box$*var–list* : *boolean–expr* :: <*stmt–list*> $\rangle$, denotes the set of statements obtained by instantiating the <*stmt–list*> with the appropriate instances of variables in the *var–list*. For example, the **assign** section of Figure 1 contains one quantified statement list, which consists of N single statements.

A single statement has two forms: simple and quantified. Examples of simple single statements are:

$x, y := y, x$	Multiple assignment: swap y and x.
$x := y \parallel y := x$	Same as $x, y := y, x$.
$x := y$ if $y \geq 0$ $\sim$ $-y$ if $y \leq 0$	Set x to absolute value of y.
$y := -y$ if $y \leq 0$	Set y to absolute value of y (identity assignment if $y > 0$).

A quantified single statement has the form $\langle \parallel$ *var–list* : *boolean–expr* :: *<stmt>* $\rangle$, where *<stmt>* is a single statement. For example, the statement $\langle \parallel i : 0 \leq i < N :: A[i] := A[i+1] \rangle$ shifts $A[1]$ to $A[0]$, $A[2]$ to $A[1]$, ..., $A[N]$ to $A[N-1]$.

UNITY is illustrated using the following problem: Sort integer array $A[0..N]$, $N \geq 0$, in ascending order.[5, p. 32] The sort program specification states that any execution of the program eventually reaches a computation state in which array element $A[i]$ does not exceed the value of element $A[i+1]$, for $i = 0, 1, \ldots, N-1$. This progress property is formalized in UNITY in the following assertion: $true \mapsto \langle \wedge i : 0 \leq i < N :: A[i] \leq A[i+1] \rangle$. Figure 1 contains a UNITY program meeting this specification.

2.4 Program Development by Composition

UNITY facilitates program development by composing a large program from many smaller programs. A large program may be composed using one of two rules, *union* and *superposition*. Software engineers have used some form of union and superposition rules for years; UNITY's contribution is a proof system by which one can deduce the properties of a composite program from its component modules.

In this paper we illustrate the use of superposition. Under superposition,

> The program is modified by adding new variables and assignments, but not altering the assignments to the original variables. Thus superposition preserves all properties of the original program. Superposition is useful in building programs in layers; variables of new layer are defined only in terms of the variables of that layer and lower ones. [5, p. 154]

A superposition is described by giving the initial values of superposed variables and transformations on the underlying program, by applying the following two rules:

Augmentation rule: A statement s in the underlying program may be transformed into a statement $s \parallel r$, where r does not assign to the underlying variables.

Restricted union rule: A statement r may be added to the underlying program provided that r does not assign values to the underlying variables.

Superposition is used in Section 7 to allow a simulation model to be specified without regard for the time flow mechanism that will be used. A particular time flow mechanism may be superposed onto an underlying simulation program.

2.5 Architecture Mappings

A mapping of a UNITY program to an architecture specifies (1) a mapping of each assignment statement to one or more processors, (2) a schedule for executing assignments (e.g., control flow), and (3) a mapping of program variables to processors.

For example, to map a UNITY program to an asynchronous shared-memory architecture, (1) above consists of partitioning the assignment statements, with each processor executing one partition. Item (2) specifies the sequence in which each processor executes the statements assigned to it. Item (3) allocates each variable to a memory module such that "all variables on the left side of each statement allocated to a processor (except subscripts of arrays) are in memories that can be written by the processor, and all variables on the right side (and all array subscripts) are in memories that can be read by the processor."[5, p. 83]

Although this mapping appears to be simple, it has a rather complex implication. A given architecture guarantees certain hardware operations to be atomic, and the programmer can only use these to build the synchronization mechanisms (e.g., locks and barriers). Meanwhile, UNITY's computational model is based on fair interleaving of atomically executed assignment statements. Therefore to obtain an efficient implementation one may need to refine the program to a more detailed level that takes into account the atomic hardware operations available on a target architecture. For example, a shared variable can be refined to be implemented by a set of variables such that the hardware atomicity corresponds to the atomicity of UNITY assignment statement execution.

3 Coupled State Transition Systems

A simulation model is represented as a *coupled state transition system*, which is a five-tuple $G = (\mathcal{S}, \Theta, V, E, \mathcal{C})$ in which:

- $\mathcal{S}$ is a set of *variables*. The set of values assumed by each variable $S \in \mathcal{S}$ is denoted by the set $\hat{S}$. Variables that assume values of simulation time are *time variables* and comprise the set Ω, all other variables are *state variables* and make up the set Σ. Note that $\mathcal{S} \equiv \Omega \bigcup \Sigma$.
- Θ is a formula specifying the initial value of one or more variables in $\mathcal{S}$.
- V is a set of graph vertices. Each vertex $v \in V$ is labeled by a variable $S \in \mathcal{S}$ and a variable value $u \in \hat{S}$. All vertices in V are uniquely labeled.
- E is a set of directed edges. The vertices joined by each edge must have identical variable labels. An edge defines a *transition* (value change) for the variable labeling the joined vertices.
- $\mathcal{C}$ is a set of *couplings*. Each coupling $c \in \mathcal{C}$ is a triple (E', T, C), where:
  - E' is a set of edges, such that $E' \subseteq E$ and the set of initial vertices for edges in E' have unique variable labels. Each edge in E belongs to exactly one coupling in $\mathcal{C}$. (A vertex may belong to more than one coupling.)
  - T is a boolean-valued function of time variables in $\mathcal{S}$.
  - C is a boolean-valued function of state variables in $\mathcal{S}$.

The functions in T take the form: $\mathbf{t} \geq$ < *time_var_exp* > where $\mathbf{t}$ is simulation time and < *time_var_exp* > is an expression involving the variables in Ω. Time variables may be assigned value through the use of random variates, written as a sequence. In random variate sequence s, the first random variate is denoted "Head(s)" and is removed by the statement, "$s :=$ Tail(s)."

The variables in $\mathcal{S}$ assume typing similar to that of traditional high level programming languages. Constants and variables that are known to be assigned a value only once are regarded as *write-once variables* and must be treated specially in the transition system (i.e. no "transition" may exist for write-once variables, only an initializing assignment may be defined.)

Some elaboration on the nature of vertex labelings is warranted. In a discrete event simulation *all* variables change values a finite number of times and therefore each value change *could* be represented by its own transition in the CSTS. However, this method is not always the most effective way to describe a system. For example, in a classical queueing model the status of the server may be enumerated as *busy* and *idle* and transitions explicitly constructed to illustrate the conditions under which ther server changes its status from one value to the other. But value changes in other model variables, for example the number

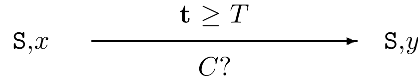

Figure 3: Portion of CSTS Illustrating One Edge, a Time Formula, and a Condition.

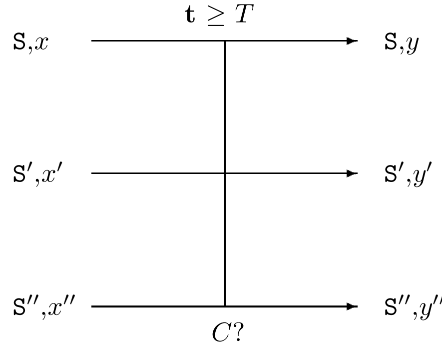

Figure 4: Portion of CSTS Illustrating Three Coupled Edges.

served by the server, might best be described as a function – i.e. under a given model condition the variable `number_served` goes from some value N to the value $N + 1$. We can represent this transition in a CSTS simply by labeling the tail of the transition by the variable `number_served` and the *value* N and the head by `number_served` and $N + 1$ (or alternately `number_served + 1`. We discuss the implications of this later.) We assume that N will be instantiated as a value in the range of valid values for `number_served`. Note that if the value of `number_served` is initially set anywhere in the specification, and changes only by the increment described above, we may be able to prove whether or not it will assume values within a specified range (we may not be able to show that it doesn't grow too large in some circumstances). Obviously, if the value `number_served` is input to the model we can provide no such proof.

The CSTS may be used to model systems with both integer and real-valued variables that assume values from either finite or infinite ranges. The only requirement is that transitions between variable values be well-defined. (Note that this requirement seems necessary for *any* specification of a discrete event simulation model.) Transitions which are labeled with specific values are called *enumerated* transitions. Transitions labeled with placeholders are *generic* transitions. Each transition type requires different treatment within the methodology.

When multiple generic transitions compose a single coupling, use of the same placeholder indicates equivalent values of the variables labeling the generic transitions. This connotation impacts the proofs for the specification. See Section 4 on generating couplings and Section 6 on generating assertions for further details on transition labeling and typing.

Figures 3 and 4 illustrate portions of CSTS's. Each figure illustrates a coupling (E', T, C) where E' contains, respectively, one and three edges. The meaning of Figure 3 is that if the value of variable `S` is x and and the simulation time is equal or greater than the time value contained in T, and if formula C is true, then `S` is assigned the value y. The meaning of Figure 4 is that if the value of variables `S,S`$'$ and `S`$''$ are x, x', and x'', respectively, and if simulation time is equal or greater than the time value contained in T, and if formula C is true, then `S,S`$'$, and `S`$''$ *simultaneously* assume the respective values y, y', and y''. Note our convention is for variables to appear in `typewriter` font. Variable values appear in either regular or italic fonts where needed for clarity of presentation.

Two graph edges are *coupled* if they belong to the same coupling. Coupled edges are denoted graphically by drawing a path consisting of undirected edges whose endpoints lie on the edges that are coupled; this is illustrated by the vertical line in Figure 4.

A simple example. We use the classical machine interference problem to illustrate CSTS concepts [7]. In the problem, a set of N semi-automatic machines fail intermittently and are repaired by one technician. 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.

For this example we consider the patrolling repairman service discipline, in which a *single* technician services all machines [15, p. 60]. In this problem, hereafter referred to as the *machine repairman problem,* (MRP) the technician traverses a path amongst the machines in a cyclic fashion $(0, 1, \ldots, N-1, 0, 1, \ldots)$. 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 variables required to model the MRP are described below. Let m denote an integer in the interval $[0, N)$ and represent machine numbers. We describe the variables that relate to the machines and the technician. We assume `N, T` and `max_repairs` are "global" simulation variables.

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

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

Therefore the value of `m.state` is *up*, *inrepair*, or *down* if m is up, in-repair, or down, respectively. Each machine also has the state variables $\texttt{m.}\lambda$, the machine failure rate, and $\texttt{m.}\mu$, the repair rate for the machine. The failure time for a machine is denoted by the time variable, `m.fail_time`. The simulation time that the repairman arrives at a machine is represented by `m.arrive_time` and the time that a repair ends is given in `m.endrepair_time.`

Technician: The location of the technician assumes one of $2N$ values: at machine 0, leaving machine 0, at machine 1, leaving machine 1, ..., at machine $N-1$, and leaving machine $N-1$.

To represent these $2N$ states, we associate with the technician a state variable `loc`, that takes on the $2N$ values 0, 0.5, 1, 1.5, 2, 2.5, ..., $N-1$, $N-0.5$, respectively. For convenience we employ boolean variables `m.a` and `m.l`, defined as:

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

Therefore the value of `loc` is 0 if the technician is at machine 0, the value is 0.5 if the technician is traveling from machine 0 to 1, the value is 1 if the technician is at machine 1, and so on. The symbols $\oplus$ and $\ominus$ are occasionally used in reference to `loc`; they denote addition and subtraction modulo N.

The state variable `num_repairs` counts the number of repairs completed by the technician.

CSTS: A simulation model of the MRP is represented by the CSTS $G_{MRP} = (\mathcal{S}, \Theta, V, E, \mathcal{C})$ in which:

- $\mathcal{S} = \{\, \mathbf{t}, \texttt{N, T, max_repairs} \,\} \cup \{\, \forall \texttt{m} : 0 \le \texttt{m} < \texttt{N} :: \texttt{m.state}, \texttt{m.}\lambda, \texttt{m.}\mu, \texttt{m.fail_time, m.arrive_time}, \texttt{m.endrepair_time} \,\} \cup \{\, \texttt{loc, num_repairs} \,\}$,
- $\Theta = \langle \forall\ \texttt{m} : 0 \le \texttt{m} < \texttt{N} :: \texttt{m.u}, \texttt{m.end_repair} = \infty, \texttt{m.fail_time} = \mathbf{t} + \mathrm{Head}(\texttt{m.}\lambda), \texttt{m.arrive_time} = \infty \,\rangle \wedge \mathbf{t} = 0 \wedge \texttt{num_repairs} = 0 \wedge \texttt{0.l} \wedge \texttt{1.arrive_time} = \mathbf{t} + \texttt{T}$ (Initially, all machines are up with first failures scheduled. The technician is leaving machine zero and an arrival time is scheduled for machine 1. All other machines have no pending arrival (arrival times set to ∞). The simulation clock is set to zero. The number of repairs is also set to zero.)

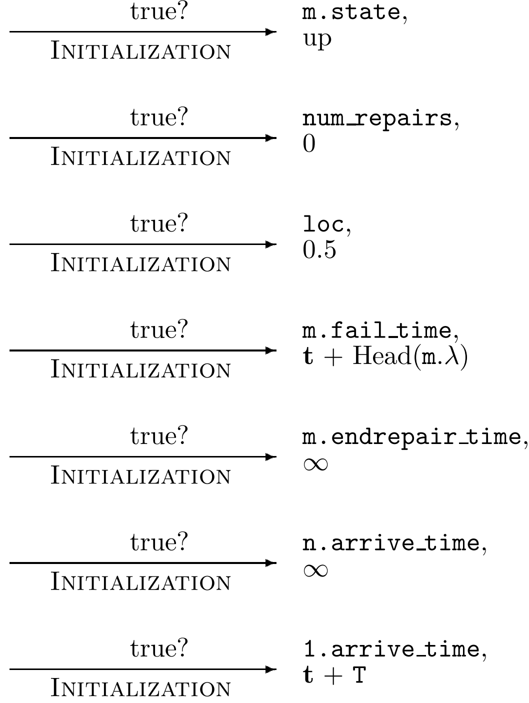

Figure 5: CSTS for MRP Initializing Transitions.

- V consists of $24N + 6$ vertices, representing all labelings of variable and variable value pairs.
- E consists of $12N + 3$ arcs, which are specified in Figures 5 through 7. Figure 5 represents $4N + 2$ arcs in the CSTS. Note that in these figures `m` is on the range $0 \leq \mathtt{m} < \mathtt{N}$ and `n` is quantified over the same range but excludes the value 1. (The initial arrive times for all machines are set to ∞ except for machine 1.) Figure 6 represents $5N + 1$ arcs and Figure 7 conveys $3N$ of the arcs in E.
- $\mathcal{C}$ consists of the $5N + 1$ couplings $\{c_i | 0 \leq i \leq 5N\}$.
  - The $4N + 2$ arcs in Figure 5 represent a single coupling based on the condition INITIALIZATION.
  - The transitions from `m.a` to `m.l` are coupled with the $\mathtt{m} \oplus \mathtt{1.arrive_time}$ transitions. (N couplings.)
  - The transitions form `m.d` to `m.i` are coupled with the `m.endrepair_time` transitions. (N couplings.)
  - The transitions from `m.i` to `m.u` are coupled with both the `m.fail_time` and `num_repairs` transitions. (N couplings.)
  - N couplings for the transitions from `m.l` to $\mathtt{m} \oplus \mathtt{1.a}$.
  - N couplings for the transitions from `m.u` to `m.d`.

Figure 5 illustrates the initializing transitions for the model. Initializing transitions occur for all variables assigned at model initialization and for all model write-once variables. An initializing transition has no value prescribed on the tail of the arc. For this example all conditions on initializing transitions are the same, i.e. the special condition INITIALIZATION which is assumed to be true only at model start-up. Figure 6 illustrates the model transitions for state variables and Figure 7 shows the model's time variable transitions.

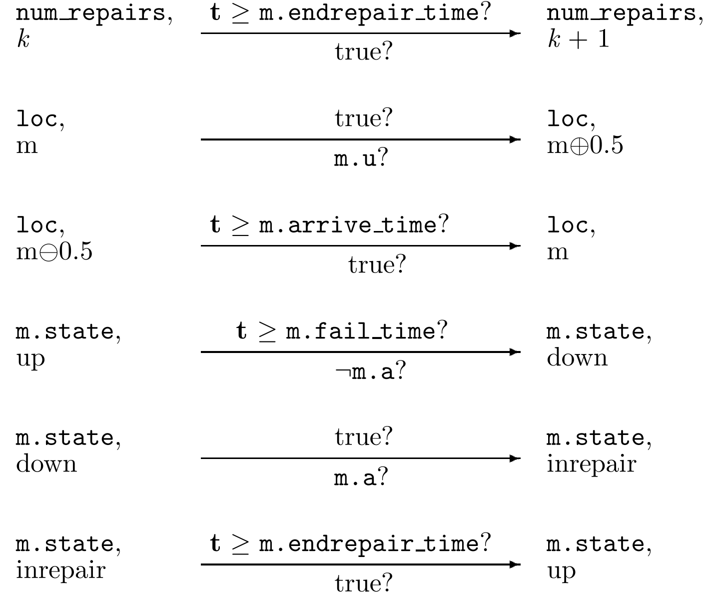

Figure 6: CSTS for MRP State Variable Transitions.

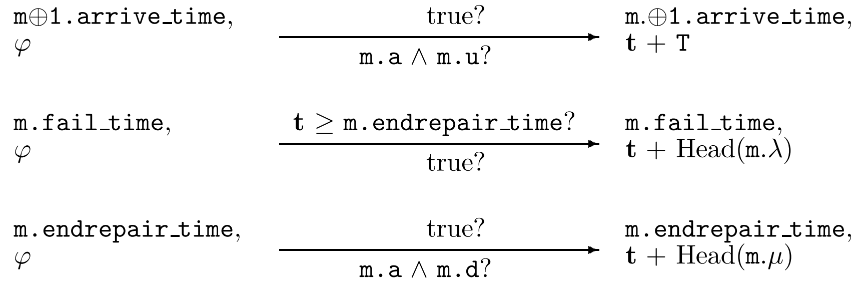

Figure 7: CSTS for MRP Time Variable Transitions.

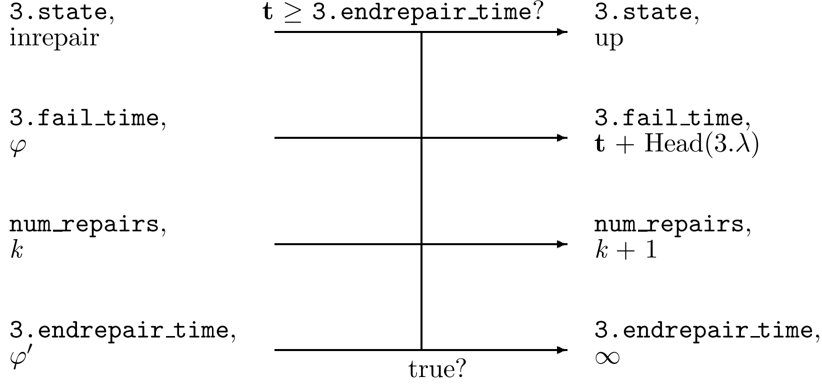

Figure 8: The Coupling Involving the Transition from *inrepair* to *up* for Machine 3.

4 Generating Couplings in a CSTS

Couplings are generated from transitions by combining all transitions with equivalent time and state conditions. Note that if a state condition for transition i requires the value specified on the tail of transition j, the two transitions can be coupled. The transitions for `m.a` to `m.l` (the second rule in Figure 6) and `m`$\oplus$`1.arrive_time` (the first rule in Figure 7) are an example of this.

Figure 8 presents the coupling involving the transition of the state variable `state` from the value *inrepair* to *up* for machine 3. We observe that this transition occurs when simulation time is greater than or equal to the value stored in `3.endrepair_time`. Three other transitions occur simultaneously with the change in value of `3.state`: the subsequent fail time for machine 3 is scheduled, the number of repairs completed is incremented by one and the value of `3.endrepair_time` is set to undefined (or ∞).

Note that we (automatically) introduce a transition to ∞ on every coupling in which a time variable is part of the condition on transitions. Setting a time value of a variable is natural when the time is that of a state change of interest to the modeler. This resetting to ∞ of a time-valued variable is needed only to facilitate correctness in the reasoning system (and potentially an eventual implementation of the model). The modeler may assume this takes place and that the placeholder generated will be unique within the coupling.

The five non-initializing couplings generated for the MRP are illustrated in Figure 9.

5 Equivalence Between CSTS and UNITY Program

Defined below is an equivalence between CSTS's and UNITY programs. CSTS $G = (\mathcal{S}, \Theta, V, E, \mathcal{C})$ is equivalent to UNITY program P. Let $L = \parallel \mathcal{C} \parallel$, and let $\mathcal{C} = \{c_1, c_2, \ldots, c_L\}$. By convention we regard c_1 as the coupling representing initialization. The program equivalent to G is defined below.

- The **constant** section: defines model constants.
- The **declare** section: declares, for each $S \in \mathcal{S}$, variable `S` with type appropriate for values in $\hat{S}$
- The **initially** section satisfies Θ and is comprised of UNITY initially statements (same as assignment except = replaces :=) for each arc in the Initialization coupling, c_1.
- The **assign** section contains, for each transition coupling $c_j = (\{e_1, e_2, \ldots e_r\}, T, C)$, for $2 \leq j \leq L$, a UNITY assignment statement. Let the initial vertex of e_i, for $1 \leq i \leq r$, be labeled by S_i, x_i and the final vertex labeled by S_i, y_i. Generate an assignment statement of the form:

$\Box \quad \langle \parallel i : 1 \leq i \leq r :: S_i := y_i \qquad \text{if } \langle S_i = x_i \wedge T \wedge C \rangle \rangle$

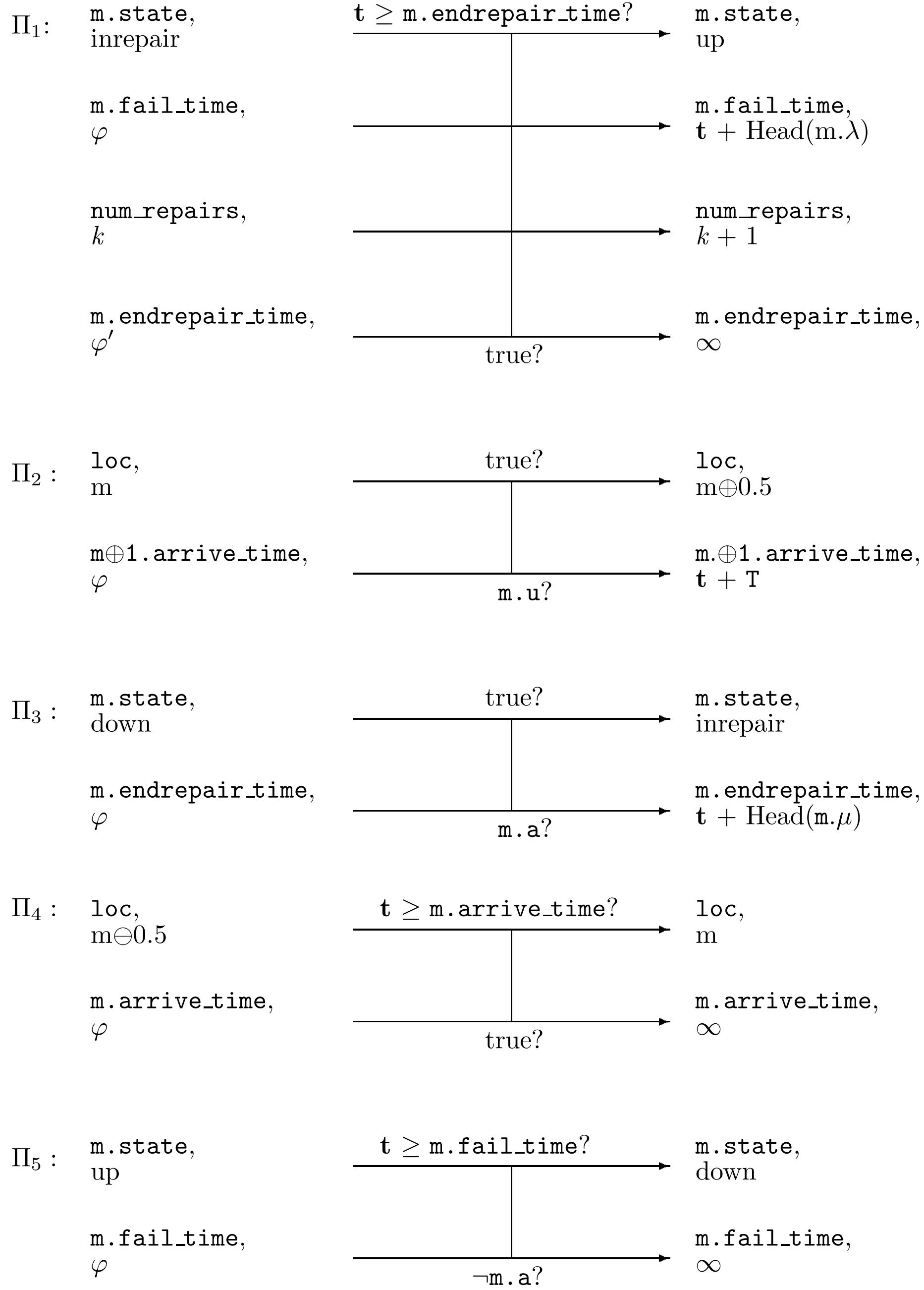

Figure 9: Non-Initializing Couplings for the MRP. Π_1 = end of repair; Π_2 = technician at an up machine; Π_3 = technician at a down machine; Π_4 = arrival; Π_5 = failure.

The UNITY program in Figure 10 is equivalent to the CSTS of Figures 5 through 7. Sequence `m.`λ (respectively, `m.`μ) is implemented by array element λ[m] (μ[m]).

Note that when generating assignments from generic transitions, if a placeholder appears in the expression on the head of the arc, and it is the same placeholder that appears on the tail of the arc then replace the placeholder by the variable name in the assignment expression. For example, if the `number_served` in the queueing model is defined by a transition going from N to $N+1$, replace the N in the expression $N+1$ by `number_served` to generate correct UNITY code. If a placeholder appears on the head of a transition arc that is not the placeholder designated at the tail, then the specification of the transition is considered ambiguous and in error.

6 Mapping CSTS to UNITY Assertions

This section describes formalization of a CSTS $G = (\mathcal{S}, \Theta, V, E, \mathcal{C})$ as a set of UNITY assertions. Generation of assertions permits the UNITY proof system to be used to establish the correctness of the assertions, and by extension, G. The specification is constructed in a two step process:

1. Formulate *unless* and *leads-to* conjectures according to the rules given below.
2. Prove the conjectures using the UNITY program generated from G as described in Section 5.

The final specification consists of those *unless* and *leads-to* conjectures whose proof succeeds and an assertion of the form, "Initial condition $\Rightarrow \Theta$." We use the notation $\overset{?}{\textit{unless}}$ and $\overset{?}{\mapsto}$ to denote conjectures, while *unless* and $\mapsto$ denote proven assertions.

Two sets of rules generate UNITY assertions; each generates assertions based on couplings in the CSTS G. The first set of rules generates assertions for couplings in which every vertex is a member of only one coupling (we refer to these as *independent* couplings). When a vertex participates in multiple couplings, a second rule provides the assertions for those *bound* couplings.

6.1 Mapping rules for independent couplings

For each independent coupling $\mathcal{C}_i = (E'_i, T_i, C_i)$ in G, generate two *unless* conjectures and one *leads-to* conjecture as follows. Let the initial vertices of $\mathcal{C}_i$ be labeled by (S_j, x_j) and the corresponding final vertices by (S_j, y_j).

$$C_i \wedge T_i \wedge \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = x_j \right) \overset{?}{\textit{unless}} \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = y_j \right) \tag{1}$$

$$C_i \wedge T_i \wedge \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = x_j \right) \overset{?}{\mapsto} \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = y_j \right) \tag{2}$$

$$\neg(C_i \wedge T_i) \wedge \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = x_j \right) \overset{?}{\textit{unless}}\ C_i \wedge T_i \wedge \left(\bigwedge_{j:1\le j\le \|E'_i\|} S_j = x_j \right) \tag{3}$$

The first assertion (Rule 1) hypothesizes that if the values of the variables S_j are x_j for all values of j, and the conditions C_i and T_i are true, then after the next value change of any variable in the simulation model, the values of S_j are either still x_j (and the conditions C_i and T_i are still true) or the values of S_j are y_j. The second assertion (Rule 2) hypothesizes that if the value of variables S_j are x_j, and the conditions C_i and T_i are true, then eventually the S_j's are assigned the values y_j. Rule 3 hypothesizes that if the values of variables S_j are x_j and either C_i or T_i (or both) are false, then the values remain x_j at least until the conditions C_i and T_i both become true.

We use these rules to generate assertions for all couplings in the MRP.

```
program MRP
  constant N=...; MaxRepairs=...; T=...; λ[1..N]=...; μ[1..N]=...;
  declare
    t                      : float
    loc                    : (0.0,0.5,1.0,...,N-1.0,N-0.5)
    state[0..N-1]          : (up, inrepair, down)
    num_repairs            : integer
    endrepair_time[0..N-1] : float
    arrive_time[0..N-1]    : float
    fail_time[0..N-1]      : float

  initially
    || t = 0.0                                   { Clock is set to zero }
    || loc = 0.5                                 { technician leaving machine zero}
    || num_repairs = 0                           { number of repairs is zero}
    || ⟨ || m :: state[m]=up ⟩                   { all machines up }
    || ⟨ || m :: fail_time[m] = t + Head(λ[m]) ⟩ { initial machine failures scheduled }
    || ⟨ || m :: endrepair_time[m] = ∞ ⟩         { no end repairs scheduled }
    || ⟨ || m :: arrive_time[m] = t + T   if m = 1 ˜ { arrive time set for machine 1; others undefined }
                                  ∞         if m ≠ 1 ⟩


  assign

  {Implementation of Π1: end of repair }
    □ ⟨ || m :: state[m] := up,
                num_repairs := num_repairs + 1,
                fail_time[m] := t + Head(λ[m]),
                endrepair_time[m] := ∞                 if m.i ∧ t ≥ endrepair_time[m] ⟩

  {Implementation of Π2: at up machine }
    □ ⟨ || m :: loc, arrive_time[m] := m ⊕ 0.5, t + T   if m.a ∧ m.u ⟩

  {Implementation of Π3: at down machine }
    □ ⟨ || m :: state[m] := inrepair,
                endrepair_time[m] := t + Head(μ[m])     if m.a ∧ m.d ⟩

  {Implementation of Π4: arrival }
    □ ⟨ || m :: loc, arrive_time[m] := m, ∞             if m ⊖ 1.l ∧ t ≥ arrive_time[m] ⟩

  {Implementation of Π5: failure }
    □ ⟨ || m :: state[m], fail_time[m] := down, ∞       if m.u ∧ t ≥ fail_time[m] ∧ ¬ m.a ⟩

end { MRP }
```

Figure 10: UNITY Program for Machine Repairman Problem. Variable m is quantified over the range $0 \leq m < N$.

6.2 Mapping rules for bound couplings

When a vertex participates in more than one coupling the vertex *binds* the couplings in so far as our ability to reason about model transitions is concerned. We say that a coupling is *bound* if any vertex on the tail of some edge in the coupling belongs to one or more different couplings.

For any CSTS, G, let κ denote a set of couplings such that each coupling in the set is bound to the other couplings in the set by virtue of common vertices. We shall refer to κ as a *binding* in G. (Note that for any CSTS, G, there may be at most $L/2$ bindings.) Let K be the set containing all bindings $\{\kappa_1, \ldots, \kappa_\alpha\}$ in G. We construct K as follows: select a bound coupling, c_1, from $\mathcal{C}$ and place it in the set κ_1. Let $\{c_\beta, \ldots, c_\gamma\}$ denote couplings, other than c_1, to which the vertices in κ_1 belong. Add these couplings to κ_1. We achieve closure on the set κ_1 by repeating this process until no vertex in κ_1 belongs to a coupling not also in κ_1. If any bound couplings remain after closure on κ_1, place one in the set κ_2 and repeat the process. Continue to create and close sets κ_i until all bound couplings belong to some κ_i.

Formally, let $\kappa_i = \{ c_{i_1}, \ldots, c_{i_n} \}$ and let the initial and final vertices of edges in c_{i_j} be labeled (S_{i_j}, x_{i_j}) and (S_{i_j}, y_{i_j}) respectively. For each binding κ_i in K, generate two *unless* conjectures and one *leads-to* conjecture as follows.

$$\bigvee_{c_{i_j} \in \kappa_i} (T_{i_j} \wedge C_{i_j} \wedge S_{i_j} = x_{i_j}) \overset{?}{\textit{unless}} \bigvee_{c_{i_j} \in \kappa_i} (S_{i_j} = y_{i_j}) \tag{4}$$

$$\bigvee_{c_{i_j} \in \kappa_i} (T_{i_j} \wedge C_{i_j} \wedge S_{i_j} = x_{i_j}) \overset{?}{\mapsto} \bigvee_{c_{i_j} \in \kappa_i} (S_{i_j} = y_{i_j}) \tag{5}$$

$$\bigvee_{c_{i_j} \in \kappa_i} (\neg(T_{i_j} \wedge C_{i_j}) \wedge S_{i_j} = x_{i_j}) \overset{?}{\textit{unless}} \bigvee_{c_{i_j} \in \kappa_i} (T_{i_j} \wedge C_{i_j} \wedge S_{i_j} = x_{i_j}) \tag{6}$$

Illustrations of bound couplings are given in Figures 11 and 12. An example of the use of the rules for bound couplings is warranted. Consider the CSTS in Figure 12. There are three couplings and one binding. $\mathcal{C}_1$ contains the two edges coupled by the conditions T_1 and C_1. $\mathcal{C}_2$ contains the two edges coupled by the conditions T_2 and C_2, and $\mathcal{C}_3$ contains the two edges coupled by the conditions T_3 and C_3. Applying rule 4 generates:

$$\begin{array}{rcl} (T_1 \wedge C_1 \wedge A = x_A \wedge B = x_B) & \vee & (T_2 \wedge C_2 \wedge B = x_B \wedge C = x_C) \vee \\ (T_3 \wedge C_3 \wedge C = x_C \wedge D = x_D) & \overset{?}{\textit{unless}} & (A = y_A \wedge B = y_B) \vee \\ (B = z_B \wedge C = y_C) & \vee & (C = z_C \wedge D = y_D) \end{array}$$

Consider again the MRP CSTS (Figure 9). Figure 13 lists the *unless* conjectures corresponding to Rule 1. Figure 14 contains the *leads-to* conjectures corresponding to Rule 2, and Figure 15 contains the *unless* conjectures generated by Rule 3. Appendix B contains an example proof and some related theorems.

7 Superposing Time Flow Mechanisms

In this section we explore how different time flow mechanisms (TFMs) may be added to a UNITY program equivalent to a CSTS representing a simulation model, also called the *underlying simulation program* (e.g., Figure 10).

To specify a simulation model, the assumption that simulation time ($\mathbf{t}$) advances is sufficient. To implement a simulation, however, one must prescribe a means by which $\mathbf{t}$ increases. Further, to prove some properties we must demonstrate that the model specification permits time to advance. We consider two general categories of time flow mechanisms: fixed time increment (FTI) and time of next event (TNE). Variations of the two types of TFMs are discussed in [16]. We add time flow to UNITY specifications via superposition. To simplify the presentation, we do not address the issue of simultaneous events.[1]

[1] For an historical perspective on handling simultaneous events in simulation refer to [21]. Some issues involving simultaneous events and parallel simulation are presented in [6, 28].

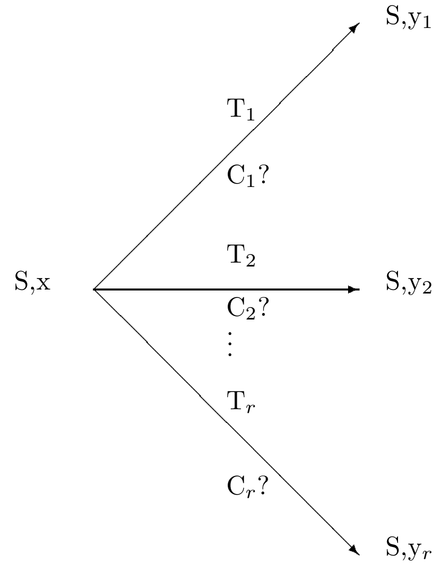

Figure 11: A Graph Containing r Bound Couplings

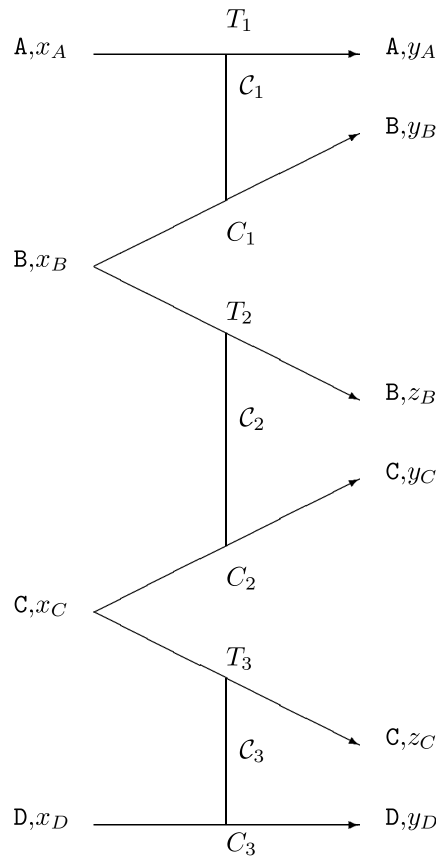

Figure 12: Graph Containing Two Bound Couplings.

$\Pi_1:$ $\mathbf{t} \geq \mathtt{m.endrepair_time} \land \mathtt{m.i} \land \mathtt{m.fail_time} = \varphi \land \mathtt{num_repairs} = \mathrm{k} \land \mathtt{m.endrepair_time} = \varphi'$
$\overset{?}{\mathit{unless}}$ $\mathtt{m.u} \land \mathtt{m.fail_time} = \mathbf{t} + \mathrm{Head}(\mathtt{m}.\lambda) \land \mathtt{num_repairs} = \mathrm{k} + 1 \land \mathtt{m.endrepair_time} = \infty$

$\Pi_2:$ $\mathtt{m.a} \land \mathtt{m.u} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.l} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \mathbf{t} + \mathtt{T}$

$\Pi_3:$ $\mathtt{m.a} \land \mathtt{m.d} \land \mathtt{m.endrepair_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.i} \land \mathtt{m.endrepair_time} = \mathbf{t} + \mathrm{Head}(\mathtt{m}.\mu)$

$\Pi_4:$ $\mathbf{t} \geq \mathtt{m.arrive_time} \land \mathtt{m}\ominus 1\mathtt{.l} \land \mathtt{m.arrive_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.a} \land \mathtt{m.arrive_time} = \infty$

$\Pi_5:$ $\mathbf{t} \geq \mathtt{m.fail_time} \land \neg\mathtt{m.a} \land \mathtt{m.u} \land \mathtt{m.fail_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.d} \land \mathtt{m.fail_time} = \infty$

Figure 13: *unless* Conjectures for MRP from Rule 1.

$\Pi_1:$ $\mathbf{t} \geq \mathtt{m.endrepair_time} \land \mathtt{m.i} \land \mathtt{m.fail_time} = \varphi \land \mathtt{num_repairs} = \mathrm{k} \land \mathtt{m.endrepair_time} = \varphi'$
$\overset{?}{\mapsto} \mathtt{m.u} \land \mathtt{m.fail_time} = \mathbf{t} + \mathrm{Head}(\mathtt{m}.\lambda) \land \mathtt{num_repairs} = \mathrm{k} + 1 \land \mathtt{m.endrepair_time} = \infty$

$\Pi_2:$ $\mathtt{m.a} \land \mathtt{m.u} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \varphi \overset{?}{\mapsto} \mathtt{m.l} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \mathbf{t} + \mathtt{T}$

$\Pi_3:$ $\mathtt{m.a} \land \mathtt{m.d} \land \mathtt{m.endrepair_time} = \varphi \overset{?}{\mapsto} \mathtt{m.i} \land \mathtt{m.endrepair_time} = \mathbf{t} + \mathrm{Head}(\mathtt{m}.\mu)$

$\Pi_4:$ $\mathbf{t} \geq \mathtt{m.arrive_time} \land \mathtt{m}\ominus 1\mathtt{.l} \land \mathtt{m.arrive_time} = \varphi \overset{?}{\mapsto} \mathtt{m.a} \land \mathtt{m.arrive_time} = \infty$

$\Pi_5:$ $\mathbf{t} \geq \mathtt{m.fail_time} \land \neg\mathtt{m.a} \land \mathtt{m.u} \land \mathtt{m.fail_time} = \varphi \overset{?}{\mapsto} \mathtt{m.d} \land \mathtt{m.fail_time} = \infty$

Figure 14: *leads-to* Conjectures for MRP from Rule 2.

$\Pi_1:$ $\mathbf{t} < \mathtt{m.endrepair_time} \land \mathtt{m.i} \land \mathtt{m.fail_time} = \varphi \land \mathtt{numrepairs\ =\ k} \land \mathtt{m.endrepair_time} = \varphi'$
$\overset{?}{\mathit{unless}}$ $\mathbf{t} \geq \mathtt{m.endrepair_time} \land \mathtt{m.i} \land \mathtt{m.fail_time} = \varphi \land \mathtt{num_repairs} = \mathrm{k} \land \mathtt{m.endrepair_time} = \varphi'$

$\Pi_2:$ $\neg\mathtt{m.u} \land \mathtt{m.a} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.u} \land \mathtt{m.a} \land \mathtt{m}\oplus 1\mathtt{.arrive_time} = \varphi$

$\Pi_3:$ $\neg\mathtt{m.a} \land \mathtt{m.d} \land \mathtt{m.endrepair_time} = \varphi \overset{?}{\mathit{unless}} \mathtt{m.a} \land \mathtt{m.d} \land \mathtt{m.endrepair_time} = \varphi$

$\Pi_4:$ $\mathbf{t} < \mathtt{m.arrive_time} \land \mathtt{m}\ominus 1\mathtt{.l} \land \mathtt{m.arrive_time} = \varphi \overset{?}{\mathit{unless}} \mathbf{t} \geq \mathtt{m.arrive_time} \land \mathtt{m}\ominus 1\mathtt{.l} \land$
$\mathtt{m.arrive_time} = \varphi$

$\Pi_5:$ $\neg(\mathbf{t} \geq \mathtt{m.fail_time} \land \neg\, \mathtt{m.a}) \land \mathtt{m.u} \land \mathtt{m.fail_time} = \varphi \overset{?}{\mathit{unless}} \mathbf{t} \geq \mathtt{m.fail_time} \land \neg\, \mathtt{m.a} \land$
$\mathtt{m.u} \land \mathtt{m.fail_time} = \varphi$

Figure 15: *unless* Conjectures for MRP from Rule 3.

program *FTI_TFM*
constant Δ=...
declare A[L-1] : integer
initially $\langle i : 1 \leq i < L :: A[i] = 0\rangle$
transform

each statement "s if b" in the underlying program to

$$s_i \text{ if } b \;\|\; A[i] := 2 \text{ if } b \wedge A[i] = 0 \sim 1 \text{ if } \neg b \wedge A[i] = 0$$

where i is the lexical statement number of s.

add to always section

atFP $= \langle \wedge i : 1 \leq i < L :: A[i] = 1\rangle$
startNewPhase $= \langle \wedge i : 1 \leq i < L :: A[i] \neq 0\rangle$

add to assign section

$\Box\langle\langle \| \; i :: 1 \leq i < L :: A[i] := 0 \quad \text{if atFP} \vee \text{startNewPhase}\rangle$
$\| \; \mathbf{t} := \mathbf{t} + \Delta \quad \text{if atFP } \rangle$

end { *FTI_TFM* }

Figure 16: Specification of Fixed Time Increment Time Flow Mechanism

7.1 Superposing fixed time increment

Let Δ denote a constant floating point value of simulation time, representing a positive nonzero time increment. The FTI algorithm consists of iterating two phases:

1. Execute all statements for the current value of $\mathbf{t}$ until the underlying simulation program reaches a fixed point (i.e., execution of any underlying program statement does not modify any variable declared in the underlying program).
2. Set $\mathbf{t}$ to $\mathbf{t} + \Delta$.

The underlying program is composed with program FTI_TFM in Figure 16 using superposition (defined in Section 2.4). FTI_TFM detects when the underlying simulation program reaches fixed point as follows. Recall from Section 5 that $L - 1$ denotes the number of statements in the underlying simulation program, because $L - 1$ is the number of non-initialization couplings in the CSTS ($L - 1 = 5N$ in Figure 10). Number the underlying program statements by the integers $1, 2, \ldots, L$. Add array $A[1..L - 1]$. Initially, all elements of array A are zero.

Array A partitions execution of underlying program statements into a set of phases such that every statement is executed at least once in each phase. Array A is initialized to zeroes each time a phase starts. The phase completes when all elements of A are nonzero. At the completion of a phase, each element of array A indicates what happened during execution of the corresponding program statement. $A[i]$ is 1 if the statement execution was the identity assignment, and 2 otherwise. If array A contains all one's, then the underlying program is in fixed point; this is the condition for a phase to end and for $\mathbf{t}$ to advance.

7.2 Superposing time-of-next-event

As with FTI, TNE requires detecting when the underlying program reaches fixed point before advancing $\mathbf{t}$. The difference between the two TFM implementations is that in FTI $\mathbf{t}$ is incremented by a fixed value and in TNE $\mathbf{t}$ is incremented to the time value of the most imminent model state change, that is the minimum value contained in all model time variables. The superposition for TNE is obtained by changing the name "FTI_TFM" to "TNE_TFM," removing the constant declaration of Δ, and the assignment "$t := t + \Delta$" to "$t := \langle \min :: \langle \textit{time_var_list}\rangle\rangle$" in Figure 16. The superposed program MRP_TNE is given in Figure 17.

```
program MRP
  constant N=...; MaxRepairs=...; T=...; λ[1..N]=...; μ[1..N]=...;
  declare
    A[5]                      : integer;
    t                         : float
    loc                       : (0.0,0.5,1.0,...,N-1.0,N-0.5)
    state[0..N-1]             : (up, inrepair, down)
    num_repairs               : integer
    endrepair_time[0..N-1]    : float
    arrive_time[0..N-1]       : float
    fail_time[0..N-1]         : float
  initially
    || ⟨i : 1 ≤ i ≤ 5 :: A[i] = 0⟩
    || t = 0.0                                      { Clock is set to zero }
    || loc = 0.5                                    { technician leaving machine zero}
    || num_repairs = 0                              { number of repairs is zero}
    || ⟨ || m :: state[m]=up ⟩                      { all machines up }
    || ⟨ || m :: fail_time[m] = t + Head(λ[m]) ⟩    { initial machine failures scheduled }
    || ⟨ || m :: endrepair_time[m] = ∞ ⟩            { no end repairs scheduled }
    || ⟨ || m :: arrive_time[m] = t + T    if m = 1 ~ { arrive time set for machine 1; others undefined }
                                  ∞          if m ≠ 1 ⟩
  always
    atFP = ⟨i : 1 ≤ 1 ≤ 5 :: A[i] = 1⟩
    startNewPhase = ⟨∧i : 1 ≤ i ≤ 5 :: A[i] ≠ 0⟩
  assign

  {Implementation of Π1: end of repair }
    □ ⟨ || m :: state[m] := up,
                 num_repairs := num_repairs + 1,
                 fail_time[m] := t + Head(λ[m]),
                 endrepair_time[m] := ∞                  if t ≥ endrepair_time[m]
        || A[1] := 2,                                    if t ≥ endrepair_time ∧ A[1] = 0 ~
                   1                                     if t < endrepair_time ∧ A[1] = 0 ⟩

  {Implementation of Π2: at up machine }
    □ ⟨ || m :: loc, arrive_time[m] := m ⊕ 0.5, t + T    if m.a ∧ m.u
        || A[2] := 2,                                    if m.a ∧ m.u ∧ A[2] = 0 ~
                   1                                     if ¬(m.a ∧ m.u) ∧ A[2] = 0 ⟩

  {Implementation of Π3: at down machine }
    □ ⟨ || m :: state[m] := inrepair,
                 endrepair_time[m] := t + Head(μ[m])     if m.a ∧ m.d
        || A[3] := 2,                                    if m.a ∧ m.d ∧ A[3] = 0 ~
                   1                                     if ¬(m.a ∧ m.d) ∧ A[3] = 0 ⟩

  {Implementation of Π4: arrival }
    □ ⟨ || m :: loc, arrive_time[m] := m, ∞              if t ≥ arrive_time[m]
        || A[4] := 2,                                    if t ≥ arrive_time[m] ∧ A[4] = 0 ~
                   1                                     if t < arrive_time[m] ∧ A[4] = 0 ⟩

  {Implementation of Π5: failure }
    □ ⟨ || m :: state[m], fail_time[m] := down, ∞        if t ≥ fail_time[m] ∧ ¬ m.a
        || A[5] := 2,                                    if t ≥ fail_time[m] ∧ A[5] = 0 ~
                   1                                     if t < fail_time[m] ∧ A[5] = 0 ⟩
  {Implementation of TFM }
    □ ⟨⟨ || i : 1 ≤ i ≤ 5 :: A[i] := 0 if atFP ∨ startNewPhase ⟩
        || t := ⟨ min : 1 ≤ m ≤ N :: endrepair_time[m], arrive_time[m], fail_time[m] ⟩ ⟩

end { MRP }
```


Figure 17: UNITY Program for Machine Repairman Problem. Variable m is quantified over the range $0 \leq m < N$.

8 Mapping UNITY Simulation Programs to Computer Architectures

The problem of mapping programs to architectures in a way that minimizes the time and space required for program execution is a difficult open problem. We propose one mapping of CSTS's to a particular target architecture, to illustrate the mapping process. The graph comprising a CSTS provides a convenient form to analyze a simulation model in devising mappings. The mapping problem can be viewed as a graph clustering problem: partition the couplings in a CSTS to minimize the number of state variables read or written by multiple partitions, and assign each partition to a processor. The target architecture selected is a distributed memory architecture, in which each processor has a private memory and processors communicate by sending messages. UNITY sequences represent the communication channels over which inter-processor messages are sent. Appending to and reading or removing from a sequence corresponds to sending and receiving messages, respectively.

An *allocation graph* for CSTS $G = (\mathcal{S}, \Theta, V, E, \mathcal{C})$ consists of a set of vertices, V_A, and a set of directed arcs, A_A. Each vertex in V_A represents one or more partitions $\mathcal{C}_i \in \mathcal{C}$. A vertex in V_A is said to *represent a state variable* $S \in \mathcal{S}$ if the coupling partition represented by V_A contains a coupling whose edge set includes an edge whose initial vertex is labeled by state variable S. Each state variable $S \in \mathcal{S}$ is represented by exactly one vertex in V_A. For all $v_1, v_2 \in V_A$, A_A contains an arc directed from v_1 to v_2 if and only if a state variable represented by v_1 appears in a condition of a coupling represented by v_2. The arc is said to be *associated with* the set all state variables represented by v_1 that appear in a condition of a coupling represented by v_2.

Recall from Section 2.5 that an architecture mapping specifies (1) a mapping of each assignment statement to one or more processors, (2) a schedule for executing assignments (e.g., control flow), and (3) a mapping of program variables to processors. CSTS G is mapped to a distributed memory architecture as follows.

For (1): Assign each vertex in V_A to a processor. In particular, assigning vertex $v \in V_A$ to processor means that the processor executes the UNITY assignment statements corresponding to couplings represented by v. The issue of how many processors to use and which partitions should be mapped to the same processor affect the program efficiency. Allocate the statement that modifies **t** in program FTI_TFM or TNE_TFM to any processor.

For (2): Statements assigned to a processor are executed iteratively.

For (3): For each state variable $S \in \mathcal{S}$, assign S to the memory module private to the processor representing the vertex in V_A representing S. For each state variable S associated with an arc in A_A, add a sequence (e.g., a stream of messages) representing variable S. The sequence is initialized with the initial value of the state variable. Each time the processor assigned to the initial vertex of the arc modifies S, it appends the new value to the sequence. Each time the processor assigned to the final vertex tests a condition containing S, the processor removes all but the last value from the sequence, and replaces occurrences of S in the condition by a read of the value of S at the head of the sequence.

Assign, for all i, $tna[i]$ to the processor which is assigned coupling c_i. Assign **t** to the processor that modifies **t**; let PR denote this processor. Add a pair of sequences, one in each direction, between PR and each processor besides PR; sequences from PR (respectively, other processors) to other processors (respectively, PR) will be referred to as *outbound* (*inbound*) sequences. Each time **t** is modified, PR appends the new value to all outbound sequences. Each time a processor other than PR modifies $tna[i]$, that processor appends the new value to the inbound sequence associated with $tna[i]$. Whenever PR modifies **t**, it first removes all but the last value from each inbound sequence, and then replaces occurrences of $tna[i]$ by the value at the head of the corresponding inbound sequence.

9 Mapping UNITY Simulation Programs to Simulation Protocols

We exemplify mapping the simulation program resulting from Section 8 to the Time Warp optimistic protocol [14]. Similar mappings can be devised for other parallel simulation protocols.

Time warp requires the time-of-next-event time flow mechanism (Section 7.2). Let pr denote the number of processors, and let the processors be numbered $PR_1, PR_2, \ldots, PR_i, \ldots, PR_{pr}$. The single variable $\mathbf{t}$ is replaced by $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_{pr}$, where $\mathbf{t}_i$ is assigned to processor PR_i. Variable $\mathbf{t}_i$ represents the local virtual time of processor PR_i, and is updated by a superposition similar to the one specified in Section 7.2, except that the superposition is done with respect to the assignment statements mapped to a single processor rather than with respect to all statements in the simulation model.

Each time a processor PR_i appends a value u to a sequence, it instead appends an ordered pair $(\mathbf{t}_i, u)$. The first value in the pair is the message timestamp used by the time warp protocol that triggers rollbacks.

Finally, it is necessary to superpose a program which will periodically save the value of all state variables in $\mathcal{S}$. This is mostly straightforward, and hence is not illustrated.

10 UNITY-Based Methodology

We next propose a simulation program development methodology using the mechanism of the preceding sections. In terms of Balci and Nance's simulation life cycle [2], assume that the "system and objectives definition" and "conceptual model" in the life cycle have been completed. We propose using a CSTS to represent the "communicative model" in the methodology. In principle it is possible to use the methodology with other formal representations of a communicative model, such as a single CSTS or a Petri net, by stating the formal semantics of the representation in UNITY (cf. Section 6) and stating rules to generate a UNITY program (cf. Section 5). The methodology itself also applies if English is used as the specification language, although one must generate the UNITY assertions and program by hand.

We propose the following methodology:

Step 1: variable definitions Define a *variable* corresponding to each simulation model attribute. Type the variable as a *time variable* if it assumes values that represent points of simulation time; otherwise type the variable as a *state variable*. Describe all transitions made between values of variables by identifying the time and state conditions under which each value change may take place. Identify the initial value of each variable. We propose that the result of this step be a CSTS as discussed in Section 3.

Verify that the CSTS matches the conceptual model. Verify that the list of constraints is complete (i.e., each arc corresponds to a valid transition, and vice versa).

Step 2: program generation Derive a UNITY simulation program from the CSTS using the rules in Section 5. The only verification required is to ensure that the rules have been properly applied.

Step 3: assertion generation Formalize the CSTS in UNITY, using the rules stated in Section 6. The only verification required is to insure that the UNITY assertions have been correctly generated.

Step 4: time flow superposition Refine the simulation program by mapping the program to a particular time flow mechanism as described in Section 7.

Overall verification of steps one to four Verify that the CSTS 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 2) implies these properties.

Step 5: architecture refinement Refine the simulation program by mapping the program resulting from Step 4 to a particular sequential or parallel computer architecture as described in Section 8.

Step 6: protocol refinement Refine the simulation program by mapping the program resulting from Step 5 to a particular sequential or parallel simulation protocol as described in Section 9.

The specification may be 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 may imply that the specification is incomplete or incorrect, or that the properties themselves do not hold for the system. Of course, Gödel's Theorem implies that some true properties may not be provable. We also observe that carrying out any such proof does not *guarantee* the correctness of the specification—you are inevitably confronted with the recursive problem of proving the Theorem prover correct—but does increase our confidence in the specification.

11 Conclusions

This paper presents a methodology to automate the construction of a simulation program from a communicative model derived from UNITY. UNITY has been illustrated to have significant potential in the development of parallel (and distributed) programs and recent work in enhancing the analyzability of fairness properties in UNITY [27] and extending UNITY to use in other domains [24] is promising. In our approach, a simulation model is specified as a coupled state transition system (CSTS), then mapped (mechanically) to a program. The program is then refined to a program suitable for a target sequential or parallel computer architecture. The refinement can be done mechanically, but further optimization by hand may be required to obtain a suitably efficient implementation. The methodology addresses formal verification as follows. A CSTS is (mechanically) mapped to a set of conjectures written as UNITY assertions. Proofs of conjectures (at present, done by hand) are carried out, and the conjectures which can be proved form the formal specification of the communicative model. The communicative model can be formally verified by stating additional properties that the CSTS should possess as UNITY assertions, and then (by hand) proving the assertion from the specification.

The proposed methodology should ultimately be incorporated into a simulation support environment which uses a higher level specification than the CSTS (e.g. an object-oriented specification) by mapping that specification to a CSTS. The nature of these higher level specification forms is an open problem. Other open problems which remain are the following:

Automating proofs. Proofs of conjectures to obtain the UNITY specification of a CSTS are generally easy to mechanize, because the proofs just require application of the rule to verify assignment statements. In contrast, proofs of properties about the specification must, at present, be done by hand. Automation is difficult because proving properties always requires identifying an order of application for UNITY theorems and sometimes requires formulation of invariants as well as metrics for induction. However, many proofs can be checked automatically using Goldschlag's system [12]. Our experience in proving the properties from the MRP and other examples 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 (not illustrated) 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*, such as the assertions generated with our approach provide. 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 to construct the formal chain of deductions required for each proof[5, Ch. 3]. 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: 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.

Reconciling proofs and graph-based analysis. Automated assistance in the verification and validation of simulation models through graph-based analysis techniques has been demonstrated, for example, in [18, 19]. Can we define the relationship between what can be proved and what can be derived graphically? Describing the reduction of the CSTS to a Simulation Graph or Petri net seems plausible. Many of the questions one would like to ask about a simulation model specification have been shown to be NP-hard or worse using other formalisms [20]. How do such limitations manifest themselves in our proof system?

Determining efficient mappings to parallel architectures and simulation protocols. Sections 8 and 9 provided one possible mapping to a distributed memory architecture and time warp. A general method of mapping the graph implied by a CSTS to a target architecture is required. Furthermore, 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. The methodology proposed here first maps a program to a time flow mechanism, then to an architecture, and finally to a simulation protocol; perhaps all three must be done jointly to obtain an optimal program in terms of execution time.

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 problem as well as 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.

Acknowledgments

The authors thank Richard E. Nance for useful discussions in writing this paper. We also thank Brad Canova and the anonymous referees for their valuable comments which improved the paper as it was developed.

References

[1] Auernheimer, B., and Kemmerer, R.A. (1986). "RT_ASLAN: A Specification Language for Real-Time Systems," *IEEE Trans. on Software Eng.,* **SE-12** (9), 879-889, September.

[2] Balci, O. (1986). "Requirements for Model Development Environments," *Computers and Operations Research,* **13** (1), 53-67.

[3] Balzer, R., and Goldman, N. (1979). "Principles of Good Software Specification and Their Implication for Specification Languages," *Proceedings of the IEEE Conf. on Specification for Reliable Software,* 58-67, April.

[4] Balzer, R., Cheatham, T.E., and Green, C. (1983). "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, **16** (11), 39-45, November.

[5] Chandy, K.M., and Misra, J. (1988). *Parallel Program Design: A Foundation,* Addison-Wesley, Reading, MA.

[6] Cota, B.A. and Sargent, R.G. (1990). "Simultaneous Events and Distributed Simulation," In: *Proceedings of the 1990 Winter Simulation Conference,* 436-440, New Orleans, LA, December.

[7] D. R. Cox and W. L. Smith (1961). *Queues*, Methuen and Company, Ltd.

[8] Derrick, E. J., Balci, O., Nance, R. E. "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," *Proc. 1989 Winter Simulation Conf.,* Wash. DC, Dec., 711-718.

[9] Dijkstra, E. (1972). "Notes on Structured Programming," *Structured Programming*, O. J. Dahl, E. Dijkstra, and C. A. R. Hoare (eds.), Academic Press.

[10] Floyd, R. W. (1976). "Assigning Meanings to Programs," *Proc. Symp. Appl. Math.*, Amer. Math. Soc., 19-32.

[11] Fujimoto, R. M. (1990). "Parallel Discrete Event Simulation," *Comm. ACM*, **33** (10), Oct., 30-53.

[12] Goldschlag, D. M. (1990). "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover," *IEEE Trans. on Software Eng.*, **16** (9), Sept., 1005-1023.

[13] Henzinger, T. A., Manna, Z., and Pnueli, A. (1990). *An Interleaving Model for Real Time*, Proc. 5th Jerusalem Conf. on Information Technology, IEEE Computer Society Press, pp. 717-730

[14] D. Jefferson (1985). "Virtual Time," *ACM Trans. on Programming Languages and Systems.*

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

[16] Nance, R. E. (1981). "The Time and State Relationships in Simulation Modeling," *Comm. ACM*, **24** (4), 173-179.

[17] Nance, R.E., and Overstreet, C.M. (1987). "Exploring the Forms of Model Diagnosis in a Simulation Support Environment," *Proceedings of the 1987 Winter Simulation Conf.,* Atlanta, GA, December 14-16, 590-596.

[18] Overstreet, C.M. (1982). *Model Specification and Analysis for Discrete Event Simulation,* Ph.D. Diss., Dept. of Computer Science, Virginia Tech, Blacksburg, VA, December.

[19] Overstreet, C.M., and Nance, R.E. (1985). "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Comm. ACM,* **28** (2), February, 190-201.

[20] Page, E.H. and Opper J.M. (1999). "Observations on the Complexity of Composable Simulation," In: *Proceedings of the 1999 Winter Simulation Conference,* pp. 553-560, Phoenix, AZ, 5-8 December 1999.

[21] Parnas, D.L. (1969). "On Simulating Networks of Parallel Processes in Which Simultaneous Events May Occur," *Communications of the ACM,* **12**(9), 519-531, September.

[22] Pnueli, A. (1981). "The Temporal Semantics of Concurrent Programs," *Theoretical Computer Science,* **13**, 45-60.

[23] Pnueli, A. and Harel, E. (1988). "Applications of Temporal Logic to the Specification of Real Time Systems," *Formal Techniques in Real-Time and Fault-Tolerant Systems,* M. Joseph (ed.), Springer-Verlag, September, 84-98.

[24] Roman, G.-C., McCann, P.J. and Plun, J.Y. (1997). "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM Transactions on Software Engineering and Methodology,* **6**(3), July, 250-282.

[25] Shankar, A. U., and Lam, S. (1992). "A Stepwise Refinement Heuristic for Protocol Construction," *ACM Trans. on Programming Languages and Systems*, **14** (3), July, 417-461.

[26] Swartout, W., and Balzer, R. (1982). "On the Inevitable Intertwining of Specification and Implementation," *Comm. ACM,* **25** (7), July, pp. 438-440.

[27] Tsai, Y.K. and Bagrodia, R.L. (1995). "Deducing Fairness Properties in UNITY Logic – A New Completeness Result," *ACM Transactions on Programming Languages and Systems,* **17**(1), January, 16-27.

[28] Wieland, F. (1997). “The Threshold of Event Simultaneity,” In: *Proceedings of the 11th Workshop on Parallel and Distributed Simulation,* 56-59, Lockenhaus, Germany, June.

[29] Zeigler, B.P. (1976). *Theory of Modelling and Simulation,* John Wiley and Sons, New York, NY.

[30] Zeigler, B.P. (1984). *Multifaceted Modelling and Discrete Event Simulation,* Academic Press, Orlando, FL.

[31] Zeigler, B.P. (1990). *Object-Oriented Simulation and Hierarchical, Modular Models,* Academic Press, Orlando, FL.

A Notation

$c, c_i, c_{i,m}^{loc}, c_{i,m}^{state}$	coupling
C, C_i	condition
$\mathcal{C}$	set of couplings
$\textsc{Cond}(\mathcal{C}_i)$	$\{C \mid (E', T, C) \in \mathcal{C}_i\}$
e_i	edge
E, E', E'_i	set of edges in coupled state transition system
E_A	set of edges in allocation graph
ES_z	set of all execution sequences of a program with time formulas equal to zero
ES_a	set of all execution sequences of a program with arbitrary time formulas
G	coupled state transition system
loc	state variable denoting location of technician
L	$\|\|\mathcal{C}\|\|$
M	number of partitions of coupling set $\mathcal{C}$ in rule II
ME	metric used in UNITY induction proof
m	machine
$m.state$	state variable denoting state of machine m
$m.d, m.i, m.u$	predicates denoting $m.state$=*down*, *inrepair*, and *up*, respectively
$m.a, m.l$	predicates denoting $loc = m$ and $loc = m \oplus 0.5$, respectively
N	number of machines in machine repairman problem
p, q, q'	assertion
P, Q, R, S, S_i	state variable
P_G	program equivalent to coupled state transition system G
pr	number of processors
PR, PR_i	processor
r	number of edges in a coupling
s, s_i	statement
$\hat{S}$	domain of state variable S
$\mathcal{S}$	set of state variables
T, T_i	time formula
$\mathbf{t}$	program variable containing current simulation time
$tna[i]$	simulation time of next assignment corresponding to coupling i
$u, w, w', x, x', x'', x_i, y, y', y'', y_i, z, z'$	state variable value
v, v_i	vertices
V	set of vertices in coupled state transition system
V_A	set of vertices in allocation graph
W	any set
Δ	simulation time increment for fixed time increment time flow mechanism
θ	initial value formula
λ	machine failure rate
μ	machine repair time

B Applying the UNITY Proof System

AXIOM 1. All time variables are set with non-zero increments to **t.**

Axiom 1 proscribes one event executing at a given time, t, from scheduling another event at that same time. Our assertion is that any model so formulated can be modified to make this a single event.

Therefore if any coupling that contains a time condition (i.e. the transitions comprising the coupling are fully or partially *determined*) is enabled at time t, executing the coupling (performing the transitions described by the coupling) *cannot* enable any other coupling (including itself) containing a time variable in the condition for the same time t; it may only enable state-based (*contingent*) couplings

Definitions:

1. We say that a coupling is *enabled* if the boolean-valued functions T and C labeling its transitions are both true.
2. A coupling is *executed* if the transitions defined by the coupling are taken.
3. If executing a coupling enables another coupling, this is a *sequence of enabled couplings* of length two.

Theorem 1: In the MRP there does not exist an infinite length sequence of enabled couplings.

Proof (by contradiction). Suppose there exists an infinite length sequence of enabled couplings in the MRP. By Axiom 1 this sequence is composed of couplings containing only state conditions. Within the MRP, the only contingent couplings are Π_2 (at an up machine) and Π_3 (at a down machine). Therefore this infinite length sequence must take the form:

$$\ldots, \Pi_2, \Pi_3, \Pi_2, \Pi_3, \ldots$$

Executing Π_2 causes m.l to hold. The condition on Π_3 is m.a. Therefore the sequence $\Pi_2, \Pi_3, \Pi_2, \ldots$ cannot be generated in the MRP. This is a contradiction of our assumption. Therefore, no infinite length sequence of enabled couplings exists in the MRP. □

Theorem 2: In the MRP_{TNE} $\mathbf{t}$ assumes all finite values of any time variable.

Proof. Let v be a time variable in the MRP_{TNE}. Assume v is set at time t to the value τ. By Axiom 1 $\tau > \mathbf{t}$. By Theorem 1 there is no infinite length sequence of enabled couplings in MRP. Therefore the conditions for updating $\mathbf{t}$ are always met after some finite number of coupling executions. If τ is the minimum of all values in all time variables of MRP, the proof is complete. If not, then by Theorem 1 $\mathbf{t}$ will again be updated and will eventually assume the value τ. □

Theorem 3: In the MRP_{FTI} $\mathbf{t}$ advances beyond all finite values of any time variable.

Proof. Let v be a time variable in the MRP_{FTI}. Assume v is set at time t to the value τ. By Axiom 1 $\tau > \mathbf{t}$. By Theorem 1 there is no infinite length sequence of enabled couplings in MRP. Therefore the conditions for updating $\mathbf{t}$ are always met after some finite number of coupling executions. If τ is less than $t + \Delta$, the proof is complete. If not, then by Theorem 1 $\mathbf{t}$ will again be updated and will eventually assume or surpass the value τ. □

Caveat. While it is *theoretically* possible that infinitely many assignments to time variables subsequent to the assignment of $v = \tau$ could generate values less than τ and thus $\mathbf{t}$ would never assume or surpass τ. This is a *practical* impossibility due to the resolution of a real valued simulation clock on any digital computer.

Deductions. The following states are mutually exclusive:

1. m.u, m.d, m.i
2. m.a, m.l

The following can be deduced from the couplings:

3. m.u $\Leftrightarrow$ m.endrepair_time $= \infty \Leftrightarrow$ m.fail_time $< \infty$
4. m.d $\Leftrightarrow$ m.fail_time $= \infty$
5. m.i $\Leftrightarrow$ m.endrepair_time $< \infty$
6. m.l $\Leftrightarrow$ m$\oplus$1.arrive_time $< \infty$
7. m.a $\Leftrightarrow$ m.arrive_time $= \infty$
8. m.i $\Leftrightarrow$ m.a

Assertion:

$\mathbf{t} \geq$ endrepair_time $\wedge$ m.i $\wedge$ m.fail_time $= \varphi \wedge$ num_repairs $=$ k $\wedge$ m.endrepair_time $= \varphi'$
unless m.u $\wedge$ m.fail_time $= \mathbf{t} +$ Head(m.λ) $\wedge$ num_repairs $=$ k $+ 1 \wedge$ m.endrepair_time $= \infty$

Proof:

Let Φ denote

$\mathbf{t} \geq$ endrepair_time $\wedge$ m.i $\wedge$ m.fail_time $= \varphi \wedge$ num_repairs $=$ k $\wedge$ m.endrepair_time $= \varphi'$

Let Φ' denote

m.u $\wedge$ m.fail_time $= \mathbf{t} +$ Head(m.λ) $\wedge$ num_repairs $=$ k $+ 1 \wedge$ m.endrepair_time $= \infty$

Show $\langle \forall s : s \text{ in } MRP_{TNE} :: \{\Phi \wedge \Phi'\}\ s\{\Phi \wedge \Phi'\}\rangle$

Consider any program state in which Φ holds.

Executing statement Π_1: causes Φ' to hold.

Executing statement Π_2: is a no-op since m.i $\Leftrightarrow \neg$ m.u (Deduction 1).

Executing statement Π_3: is a no-op since m.i $\Leftrightarrow \neg$ m.d (Deduction 1).

Executing statement Π_4: is a no-op since m.i $\Leftrightarrow$ m.a $\Leftrightarrow$ m.arrive_time $= \infty$ (Deductions 7,8).

Executing statement Π_5: is a no-op since m.i $\Leftrightarrow \neg$ m.u $\Leftrightarrow$ m.fail_time $= \infty$ (Deductions 1,3).

Executing statement TFM: has no effect on Φ.

Therefore the assertion holds.

Further since only Π_1 establishes Φ', Φ *ensures* Φ' and by implication, $\Phi \mapsto \Phi'$. □