Formally Reasoning About and Automatically Generating Sequential and Parallel Simulations

Marc Abrams and Ernest H. Page

TR 92-55

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

December 19, 1992

Formally Reasoning About and Automatically Generating Sequential and Parallel Simulations

Marc Abrams
Ernest H. Page

Department of Computer Science
Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
{abrams,page}@vtopus.cs.vt.edu

Computer Science Department TR-92-55
Systems Research Center SRC-92-009

December 19, 1992

Abstract

This paper proposes a methodology to automate the construction of simulation programs within the context of a simulation support environment. The methodology starts with a simulation model specification in the form of a set of coupled state transition systems. The paper provides a mechanical method of mapping the transition systems first into a set of formal assertions, permitting formal verification of the transition systems, and second into an executable program. UNITY, a computational model and proof system suitable for development of parallel and distributed programs through step-wise refinement of specifications, is used as the specification and program notation. 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

Contents

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

1 Introduction

Model representation, a critical process within the simulation model life cycle, involves the translation of a *conceptual model* (the model that exists in the mind of the modeler) into one or more *communicative models* (models that can be communicated to others) [2, p. 57]. Model representation, commonly referred to as *model specification*, is designed to enunciate *what* a model is to do as separate from *how* it is to be done [3] – to the greatest extent possible [10, 31].

The specification process is typically realized using a *specification language*. A myriad of specification languages have been proposed for system development. Examples include AXIS [14], PSL/PSA [32], PDL [5], RDL [15], JSD [6, 18], Entity-Life Modeling [29], and PAISLey [33, 34, 35]. Several simulation-specific specification languages have also appeared, among them, DELTA [17], GEST [24], ROSS [20], and the Condition Specification [25, 26].

Many extant software and simulation specification languages are formal in nature. Precision in language syntax and semantics facilitates diagnostic analysis of specifications to assist in the assessment of, among other things, model correctness and completeness [23, 26]. In addition, formal reasoning methods are required to realize fully the automation-based paradigm [4] through a simulation support environment.

Although methods for formal reasoning about sequential, general purpose computer programs were first proposed twenty-five years ago [11], few methods permit formal reasoning about simulation model specifications. Simulation requires methods for reasoning about: (1) simulation time, (2) prioritization of events or actions scheduled for the same simulation time, and (3) output measures. While problems (1) and (2) are addressed in the areas of real-time systems [1, 28] and communication protocols [30, Section 6.1], no existing methods address problem (3). In addition, a formal reasoning system for simulation must support model development using any conceptual framework [9], time flow mechanism, sequential or parallel execution protocol [12], and target computer architecture.

This paper is a first step towards developing a formal reasoning system for simulation. We propose a methodology to automate construction and verification of sequential and parallel simulation programs from a communicative model using the UNITY computation model and proof system [7], introduced in Section 2. The proposed methodology supports multiple time flow mechanisms, execution protocols, and target computer architectures. Within the methodology, 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 definition of CSTS requires the domain of each state variable be discrete. 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 mechanically maps the CSTS to a set of UNITY assertions using the rules of Section 5. A novel aspect of our methodology is that a simulation modeler can state properties that a correct communicative model must possess, and then use the UNITY proof system to formally verify the correctness of the CSTS. The third step mechanically maps the CSTS to a UNITY program, using the rules of Section 4. Use of the rules guarantees that the program meets the specification embodied by the CSTS. The fourth step maps the UNITY simulation program to a time flow mechanism. A mechanical method of mapping to fixed time increment and time-of-next-event is given in Section 6. The fifth step maps the program resulting from step four to a target computer architecture. One heuristic for this step is given in Section 7. 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 8. The methodology is summarized in Section 9, and illustrated with the machine interference problem.

2 Introduction to UNITY

Chandy and Misra's UNITY is used as a basis for the methodology for several reasons:

- 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., [27, 30]). 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 used to over-specifying control flow. In fact, efforts to automatically transform sequential FORTRAN programs to parallel programs require code analysis to identify what control flow constraints can be relaxed.

- UNITY proof rules are based on temporal logic, and real time extensions to temporal logic have been proposed (e.g., [16]), which could be used to reason about simulation time and ordering events and actions scheduled for the same simulation time.
- UNITY assertions permit algebraic as opposed to 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 [13], which is essential to achieving the automation-based paradigm.

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. 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 but 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"[7, 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.[7, 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), $+$, 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.[7, 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.

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.

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

Figure 1: UNITY 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 1.

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*> □ <*stmt*> □ ⋯ □<*stmt*>. The symbol "□" separates statements. A <*stmt*> is either a quantified statement list or a single statement. A quantified statement list, ⟨□*var-list* : *boolean-expr* :: <*stmt-list*> ⟩, 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 2 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 $y \geq 0$ ~ $-y$ 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 ⟨‖ *var-list* : *boolean-expr* :: <*stmt*> ⟩, 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.[7, 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 2 contains a UNITY program meeting this specification.

```
program sort
assign
    ⟨$\Box i : 0 \leq i < N$ :: $A[i], A[i+1] := A[i+1], A[i]$  if A[i]>A[i+1]⟩
end {sort}
```

Figure 2: Sort Array A into Ascending Order.

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*; we use only the later. 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 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. [7, 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 6 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."[7, 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 finite set of *state variables*. Each variable $S \in \mathcal{S}$ has as its domain a finite set, denoted $\hat{S}$.

- 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 state 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 state variable labels and distinct variable values.
- $\mathcal{C}$ 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 state 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 *time formula*, whose value is a floating point quantity.
  - $\mathcal{C}$ is a *condition*, which is a Boolean function of variables in $\mathcal{S}$.

Time formulas may use random variates, written as a sequence. In random variate sequent s, the first random variate is denoted "Head(s)" and is removed by the statement, "$s := Tail(\mathrm{s})$."

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 has been x for exactly T simulation time units, and if formula C is true, then S *may* be assigned the value y. The meaning of Figure 4 is that if the value of variables S, S' and S'' have been x, x', and x'', respectively, for exactly T simulation time units, and if formula C is true, then S, S', and S'' may be assigned *simultaneously* the values y, y', and y'', respectively. However, in both Figures 3 and 4, the current simulation time ($\mathbf{t}$ in Section 4) will not advance until either S is assigned y or another transition makes C false.

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.

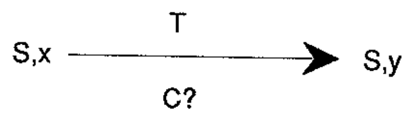

Figure 3: Portion of CSTS illustrating one edge, a time formula, and a condition.

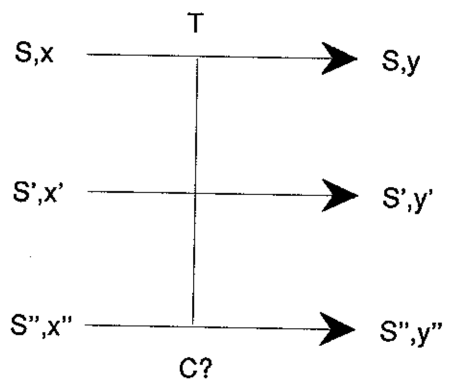

Figure 4: Portion of CSTS illustrating three coupled edges.

Example 1 The classical machine interference problem is used throughout this paper to illustrate concepts [8]. 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.

This paper considers the patrolling repairman service discipline, in which a *single* technician services all machines [21, 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 state variables required to model the MRP are described below. Let m denote an integer in the interval $[0, N)$ and represent machine numbers.

Machines: Each machine m is in one of three states: up, inrepair, or down. Associated with each 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:

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

Therefore the value of $m.state$ is *up*, *inrepair*, or *down* if m is up, in-repair, or down, respectively.

Technician: The technician is in one of $2N$ states: 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:

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

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 symbol $\oplus$ is occasionally used in reference to loc; it denotes addition modulo N.

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

- $\mathcal{S} = \{\forall m : 0 \le m < N :: \text{m.state}\} \cup \{loc\}$,
- $\Theta = \langle \forall m : 0 \le m < N :: m.u \rangle \wedge 0.l$ (initially, all machines are up and the technician is leaving machine zero, and
- V consists of $5N$ vertices, representing all labelings of state variable and variable value pairs. The $5N$ vertices arise because state variable loc has $2N$ values in its domain and each of the N variables $m.state$ takes on 3 values.
- E consists of $5N$ edges, which are specified in Figure 5. The figure illustrates five edges, but each edge is quantified over the N values of m, yielding $5N$ vertices.
- $\mathcal{C}$ consists of the $5N$ couplings $\{c_i | 0 \le i < 5N\}$. Each edge in E belongs to a unique coupling. Figure 5 gives the correspondence between couplings and edges, using the notation:

  $\langle \forall m :: c^{loc}_{1,m}$ denote $c_m \rangle$,

  $\langle \forall m :: c^{loc}_{2,m}$ denote $c_{N+m} \rangle$,

  $\langle \forall m :: c^{state}_{1,m}$ denote $c_{2N+m} \rangle$,

  $\langle \forall m :: c^{state}_{2,m}$ denote $c_{3N+m} \rangle$, and

$\langle \forall m :: c^{state}_{3,m}$ denote $c_{4N+m} \rangle$.

Therefore, couplings c_0 through c_{2N-1} change the value of *loc*, and couplings c_{2N} through c_{5N-1} change the value of $\langle \forall m :: m.state \rangle$.

In Figure 5 and all assertions in this paper, universal quantification over the values of variable m is assumed. Exactly one edge belongs to each coupling in $\mathcal{C}$. Coupling $c^{loc}_{1,m}$ requires that whenever the technician is at machine m and m is up, he leaves the machine. Coupling $c^{loc}_{2,m}$ requires that the technician arrives at machine $m \oplus 1$ exactly T simulation time units after it leaves machine m. Coupling $c^{state}_{1,m}$ requires that machine m remains up for at least Head$(m.\lambda)$ time units, after which it goes down as soon as the technician is not at machine m. ($c^{loc}_{1,m}$ and $c^{state}_{1,m}$ together imply that machine m remains up for exactly Head$(m.\lambda)$ time units.) Coupling $c^{state}_{2,m}$ requires that a down machine enters repair when the technician arrives. Coupling $c^{state}_{1,m}$ requires that a machine remains in repair exactly Head$(m.\mu)$ time units. □

4 Equivalence between CSTS's and UNITY Programs

Simulation time is assumed to be a floating point number, denoted by the type name "float," and represented by program variable $\mathbf{t}$. The program generated by the rules in this section reads $\mathbf{t}$, while additional code given in Section 6 updates $\mathbf{t}$.

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 $\mathcal{C} = \{c_1, c_2, \ldots, c_L\}$. The program equivalent to G is defined below.

- The **declare** section:
  - declares, for each $S \in \mathcal{S}$, variable S with type appropriate for domain $\hat{S}$; and
  - declares variables $tna[1], tna[2], \ldots, tna[L]$ of type float. (The name "*tna*" is a mnemonic for "[simulation] time of next assignment.")

$$\begin{tikzcd}
c^{loc}_{1,m}: & \text{loc}, \text{m.a} \arrow[r, "0"] \arrow[r, "\text{m.u?}"'] & \text{loc}, \text{m.l} \\
c^{loc}_{2,m}: & \text{loc}, \text{m.l} \arrow[r, "\text{T}"] \arrow[r, "\text{true?}"'] & \text{loc}, \text{m}\oplus 1.\text{a} \\
c^{state}_{1,m}: & \text{m.state}, \text{m.u} \arrow[r, "\text{Head}(\text{m}.\lambda)"] \arrow[r, "\neg\text{m.a?}"'] & \text{m.state}, \text{m.d} \\
c^{state}_{2,m}: & \text{m.state}, \text{m.d} \arrow[r, "0"] \arrow[r, "\text{m.a?}"'] & \text{m.state}, \text{m.i} \\
c^{state}_{3,m}: & \text{m.state}, \text{m.i} \arrow[r, "\text{Head}(\text{m}.\mu)"] \arrow[r, "\text{true?}"'] & \text{m.state}, \text{m.u}
\end{tikzcd}$$

Figure 5: CSTS for MRP. All figures are quantified over all m $(0 \leq m < N)$; therefore the graph consists of $5N$ arcs.

- The **initially** section satisfies Θ and contains:

$$\langle \forall i : 1 \leq i \leq L :: tna[i] = \infty \rangle.$$

- The **assign** section contains, for each coupling $c_j = (\{e_1, e_2, \ldots e_r\}, T, C)$, for $1 \leq j \leq L$, the following two statements. Let the initial, (respectively, final) vertex of e_i, for $1 \leq i \leq r$, be labeled by S_i, x_i (S_i, y_i). Let i be quantified over $1 \leq i \leq r$ in the following.

$$\square \quad tna[j] := \mathbf{t} + T \qquad \text{if} \langle \wedge i \ : \ 1 \leq i \leq r \ :: \ Si = x_i \rangle \wedge tna[j] = \infty$$

$$\begin{array}{lll} \square & \langle tna[j] := \infty & \text{if} \langle \wedge i \ : \ 1 \leq i \leq r \ :: \ Si = x_i \rangle \wedge tna[j] \leq \mathbf{t} \wedge C \\ & \| \ \langle \| \ i :: Si := y_i & \text{if} \langle \wedge i \ : \ 1 \leq i \leq r \ :: \ Si = x_i \rangle \wedge tna[j] \leq \mathbf{t} \wedge C \rangle \rangle \end{array}$$

A special case of the above arises if time formula T is zero; in this case the **assign** section contains one UNITY program statement:

$$\square \quad \langle \| \ i \ :: \ Si := y_i \qquad \text{if} \langle \wedge i \ : \ 1 \leq i \leq r \ :: \ Si = x_i \rangle \wedge C \rangle$$

Example 2 The UNITY program in Figure 6 is equivalent to the CSTS of Figure 5. Sequence $m.\lambda$ (respectively, $m.\mu$) is implemented by array element λ[m] (μ[m]). The **always** section equates elements of array *tna* to the symbolic names tna_loc_2, tna_state_1, and tna_state_3 to help document the code. Note that array elements *tna*[0..N-1] and *tna*[3N..4N-1] are not used because $c_{1,m}^{loc}$ and $c_{3,m}^{state}$ have time formulas of zero. Finally, note that the quantified assignment statements shown above appear as single statements (i.e., the "||" operator is omitted) because the value of r in the above formulas is one. $\square$

5 Mapping CSTS's 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 hence of G.

The specification is constructed in a two step process:

```
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)
    tna[0..5N-1]   : float

  initially
    || t=0.0
    || ⟨ i : 0 ≤ i ¡ 5N :: tna[i]=∞ ⟩                {no state transitions scheduled}
    || ⟨ || m :: state[m]=up ⟩                       { all machines up }
    || loc=0.5                                       { technician leaving machine zero}

  always
    || ⟨|| m :: tna_loc_2[m]     = tna[N+m] ⟩
    || ⟨|| m :: tna_state_1[m]   = tna[2N+m] ⟩
    || ⟨|| m :: tna_state_3[m]   = tna[4N+m] ⟩

  assign
  {Implementation of c^loc_1,m:}
    □⟨ □ m :: loc := loc ⊕ 0.5                        if loc=m ∧ state[m]=up ⟩

  {Implementation of c^loc_2,m:}
    □⟨ □ m :: tna_loc_2[m] := t + T                   if tna_loc_2[m]= ∞ ∧ loc=m⊕0.5 ⟩
    □⟨ □ m :: ⟨ loc, tna_loc_2[m] := loc ⊕ 0.5, ∞     if tna_loc_2[m]≤ t ∧ loc=m⊕0.5 ⟩⟩

  {Implementation of c^state_1,m:}
    □⟨ □ m :: tna_state_1[m] := t + Head(λ[m])        if tna_state_1[m]= ∞ ∧ state[m]=up ⟩
    □⟨ □ m ::
         ⟨ state[m], tna_state_1[m] := down, ∞        if tna_state_1[m]≤ t ∧ state[m]=up ∧ loc ≠ m ⟩⟩

  {Implementation of c^state_2,m:}
    □⟨ □ m :: state[m] := inrepair                    if state[m]=down ∧ loc=m ⟩

  {Implementation of c^state_3,m:}
    □⟨ □ m :: tna_state_3[m] := t + Head(μ[m])        if tna_state_3[m]= ∞ ∧ state[m]=inrepair ⟩
    □⟨ □ m :: ⟨state[m], tna_state_3[m] := up, ∞      if tna_state_3[m]≤ t ∧ state[m]=inrepair ⟩⟩
end { MRP }
```

Figure 6: UNITY Program for Machine Repairman Problem. Variable m is quantified over the range $0 \leq m < N$.

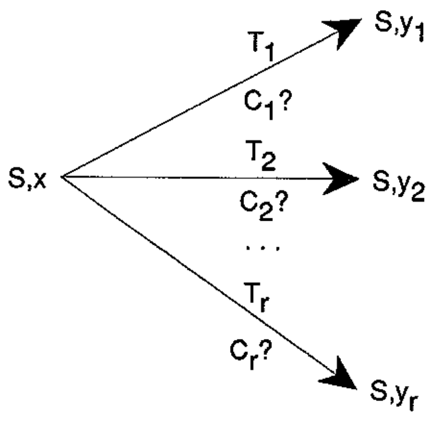

Figure 7: Illustration of mapping CSTS's to UNITY conjectures via rule I.

1. Formulate *unless* and *leads-to* conjectures according to rules I and II, below.

2. Prove the conjectures using the UNITY program equivalent of G with all time formulas equal to zero.

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 rules generate UNITY assertions. The first rule states properties about transitions out of individual vertices in G. The second rule states properties about simultaneous transitions represented by couplings.

Rule I: For each vertex $v \in V$ in G, generate two *unless* conjectures and one *leads-to* conjectures as follows. Consider the set of all edges with initial vertex v; let r denote the number of such edges, and let $\{e_1, e_2, \ldots e_r\}$ denote the edges. Let the label of vertex v be (S, x), and let the final vertex of e_i is be (S, y_i), for $1 \leq i \leq r$. The definition of coupling implies that each edge e_i belongs to a distinct coupling; let edge e_i belong to coupling (E'_i, T_i, C_i). This is illustrated in Figure 7. First, generate the conjectures:

$$S = x \;\wedge\; \left(\bigvee_{1\le i\le r} C_i\right) \;\overset{?}{unless}\; \left(\bigvee_{1\le i\le r} S = y_i\right)$$

$$S = x \;\wedge\; \left(\bigvee_{1\le i\le r} C_i\right) \overset{?}{\mapsto} \left(\bigvee_{1\le i\le r} S = y_i\right)$$

In addition, if $(\neg \bigvee_{1\le i\le r} C_i \Rightarrow true)$ (i.e., it is not the case that one of the C_i's holds at every point during the simulation), then additionally generate:

$$S = x \;\wedge\; \neg\left(\bigvee_{1\le i\le r} C_i\right) \;\overset{?}{unless}\; S = x \;\wedge\; \left(\bigvee_{1\le i\le r} C_i\right)$$

The third assertion is only generated for $c^{loc}_{1,m}$, $c^{state}_{1,m}$, and $c^{state}_{2,m}$ of Figure 5.

The first assertion hypothesizes that if the value of state variable S is x, and one of the conditions $C_1, \ldots, C_r$ is true, then after the next change in any state variable value in the simulation model, the value of S is either still x (and one of the conditions $C_1, \ldots, C_r$ is still true) or changes to one of $y_1, \ldots, y_r$. The second assertion hypothesizes that if the value of state variable S is x, and one of the conditions $C_1, \ldots, C_r$ is true, then eventually S is assigned one of $y_1, \ldots, y_r$. The third assertion hypothesizes that if the value of state variable S is x and none of the conditions $C_1, \ldots, C_r$ are satisfied, then the value remains x at least until one of the conditions $C_1, \ldots, C_r$ becomes true.

Rule II: A vertex in a CSTS often has multiple outgoing edges, which must belong to distinct couplings (by the definition of a coupling). Formulation of assertions about the effect of coupled transitions requires partitioning the couplings into M partitions, denoted $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_M$, so that all transitions out of a vertex belong to the same partition. Formally, let $\mathcal{C} = \cup_{1\le i\le M}\mathcal{C}_i$, where $c, c' \in \mathcal{C}_i$ if and only if the set of initial vertices of edges in c and c' are not disjoint and $\cap_{1\le i\le M}\mathcal{C}_i = \emptyset$. Let $\textsc{Cond}(\mathcal{C}_i)$ denote the set of conditions labeling all couplings in partition $\mathcal{C}_i$; formally $\textsc{Cond}(\mathcal{C}_i) = \{C | (E', T, C) \in \mathcal{C}_i\}$. For each partition $\mathcal{C}_i$ containing at least two couplings, generate the following conjecture:

$$\bigvee_{C\in\textsc{Cond}(\mathcal{C}_i)} \left[C \wedge \bigwedge_{S\in SV(C)} S = LV(S, C)\right] \;\overset{?}{unless}\; \bigvee_{C\in\textsc{Cond}(\mathcal{C}_i)} \left[\bigwedge_{S\in SV(C)} \bigvee_{v\in RV(S,C)} S = v\right]$$

where, for each $(E', T, C) \in \mathcal{C}$,

- $SV(C)$ denotes the set of state variables labeling initial vertices of edges in E',
- $LV(S, C)$, for each state variable S labeling an initial vertex V_S of an edge in E', denotes the variable value labeling V_S, and
- $RV(S, C)$, denotes the set of variable values labeling the final vertices of all edges in E' labeled by state variable $S \in \mathcal{S}$.

Example 3 The intention of rule II can be explained through an example. Consider the CSTS in Figure 8. There are two partitions. $\mathcal{C}_1$ contains the edge whose initial and final vertices are labeled by P, and $\mathcal{C}_2$ contains all remaining edges. $\mathcal{C}_1$ contains only one coupling, and therefore rule II generates no conjectures for $\mathcal{C}_1$. $\mathcal{C}_2$ contains three couplings, and therefore rule II generates:

$$(C_1 \wedge Q = x \wedge R = y) \vee (C_2 \wedge R = y \wedge S = z) \overset{?}{unless}$$
$$(Q = x' \wedge R = y') \vee (R = y'' \wedge S = z')$$

The second assertion states that if state variables $Q, R,$ and S have value $x, y,$ and z and one of the two coupling conditions C_1 or C_2 is true, then the three state variables retain these values either forever or until the transition implied by one set of coupled edges occurs (i.e., either Q and R are assigned x' and y', or R and S are assigned y'' and z', respectively). □

Rules I and II ignore time formulas specified in couplings, and instead use the value zero for each time formula. The UNITY computation model does not provide an explicit method to reason about time. Nevertheless, UNITY is adequate for reasoning about simulation model properties not involving time, because any property of a transition system with time formulas equal to zero is a property of a transition system that represents non-zero formulas. We leave the specification of and formal reasoning about simulation time as an open problem. □

Lemma 1 *Any property of CSTS G with all time formulas equal to zero holds for G with arbitrary time formulas.*

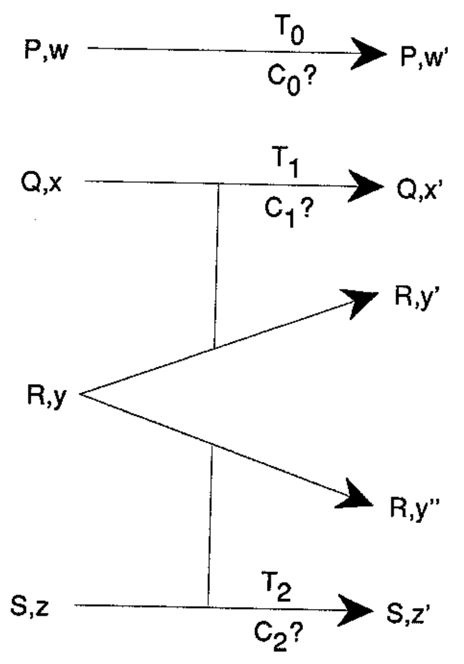

Figure 8: Illustration of mapping CSTS's to UNITY conjectures via rule II.

Proof: Consider the UNITY program equivalent of G, denoted P_G. The set of all execution sequences of P_G with all time formulas in all couplings equal to zero, denoted ES_z, is a superset of the set of all execution sequences of P_G with arbitrary time formulas, denoted ES_a. Therefore any property that holds for all execution sequences in ES_z holds for all execution sequences in ES_a.

□

Example 4 Consider again the MRP CSTS (Figure 5). Figure 9 lists the *unless* conjectures corresponding to rule I. Figure 10 contains the *leads-to* conjectures corresponding to rule I. (Rule II partitions set $\mathcal{C}$ partitioned into $5N$ partitions, each containing a single coupling of $\mathcal{C}$, however rule II generates no assertions in this example.) Appendix B contains selected proofs. The final specification (Figure 11) contains all *unless* conjectures except $m.u \wedge \neg m.a$ *unless* $m.d$, all *leads-to* conjectures except $m.u \wedge \neg m.a \mapsto m.d$, and one assertion implied by Θ. Note that the *until* relation is used to combine *unless* and *leads-to* assertions when possible in Figure 11. □

From $c^{loc}_{1,m}$: $m.a \wedge \neg m.u \overset{?}{unless}\ m.a \wedge m.u$

From $c^{loc}_{1,m}$: $m.a \wedge m.u \overset{?}{unless}\ m.l?$

From $c^{loc}_{2,m}$: $m.l \overset{?}{unless}\ m \oplus 1.a$

From $c^{state}_{1,m}$: $m.u \wedge m.a \overset{?}{unless}\ m.u \wedge \neg m.a$

From $c^{state}_{1,m}$: $m.u \wedge \neg m.a \overset{?}{unless}\ m.d*$

From $c^{state}_{2,m}$: $m.d \wedge \neg m.a \overset{?}{unless}\ m.d \wedge m.a$

From $c^{state}_{2,m}$: $m.d \wedge m.a \overset{?}{unless}\ m.i$

From $c^{state}_{1,m}$: $m.i \overset{?}{unless}\ m.u$

Figure 9: *unless* conjectures resulting from rule I. Starred conjecture does not hold.

From $c^{loc}_{1,m}$: $m.a \wedge m.u \overset{?}{\mapsto} m.l$

From $c^{loc}_{2,m}$: $m.l \overset{?}{\mapsto} m \oplus 1.a$

From $c^{state}_{1,m}$: $m.u \wedge \neg m.a \overset{?}{\mapsto} m.d*$

From $c^{state}_{2,m}$: $m.d \wedge m.a \overset{?}{\mapsto} m.i$

From $c^{state}_{1,m}$: $m.i \overset{?}{\mapsto} m.u$

Figure 10: *leads-to* conjectures resulting from CSTS. Starred conjecture cannot be proven in UNITY's proof system.

MRP1 : $m.a \wedge \neg m.u$ *unless* $m.a \wedge m.u$

MRP2 : $m.a \wedge m.u$ *until* $m.l$

MRP3 : $m.l$ *until* $(m \oplus 1).a$

MRP4 : $m.u \wedge m.a$ *unless* $m.u \wedge \neg m.a$

MRP5 : $m.d \wedge \neg m.a$ *unless* $m.d \wedge m.a$

MRP6 : $m.d \wedge m.a$ *until* $m.i$

MRP7 : $m.i$ *until* $m.u$

MRP8 : Initial condition $\Rightarrow \langle \forall m : 0 \leq m < N :: m.u \rangle \wedge 0.l$

Figure 11: Final specification of MRP.

6 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 6).

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. 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 [22]. We add time flow to UNITY specifications via superposition.

6.1 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 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 $\mathbf{t} + \Delta$.

```
program FTI_TFM
constant Δ=...
declare A[L] : integer
initially ⟨i : 1 ≤ i ≤ L :: A[i] = 0⟩
transform

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

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

    where i is the lexical statement number of s.

add to always section

    atFP = ⟨∧i : 1 ≤ i ≤ L :: A[i] = 1⟩
    startNewPhase = ⟨∧i : 1 ≤ i ≤ L :: A[i] ≠ 0⟩

add to assign section

    ⟨|| i :: 1 ≤ i ≤ L :: A[i] := 0    if atFP ∨ startNewPhase⟩
    || t := t + Δ    if atFP

end { FTI_TFM }
```

Figure 12: Specification of Fixed Time Increment Time Flow Mechanism

The underlying program is composed with program FTI_TFM in Figure 12 using superposition (defined in Section 2.4). FTI_TFM detects when the underlying simulation program reaches fixed point as follows. Recall from Section 4 that L denotes the number of statements in the underlying simulation program, because L is the number of couplings in the CSTS ($L = 5N$ in Figure 6). Number the underlying program statements by the integers $1, 2, \ldots, L$. Add array $A[1..L]$. 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.

6.2 Superposing Time-of-Next-Event

As with FTI, TNE requires detecting when the underlying program reaches fixed point before advancing **t**. The difference between the two TFM implementations is that in FTI **t** is incremented by a fixed value and in TNE **t** is incremented to the time value of the most imminent model state change, that is the minimum element of array *tna*. The superposition for TNE is obtained by changing the name "FTI_TFM" to "TNE_TFM" and the assignment "$t := t + \Delta$" to "$t := \langle \min : 1 \leq i \leq L :: tna[i] \rangle$" in Figure 12.

7 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 $\mathbf{t}$, 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.

Example 5 One allocation graph for the MRP consists of $N+1$ vertices as follows. Each of the $N+1$ state variables in $\mathcal{S}$ are represented by a unique vertex in V_A. Formally, N vertices of V_A each represent, for each value of m, $\{c_{1,m}^{state}, c_{2,m}^{state}, c_{3,m}^{state}\}$, and the remaining vertex of V_A represents $\{c_{1,m}^{loc}, c_{2,m}^{loc}\}$.

The mapping to a distributed memory architecture described above can use $N+2$ processors, one for each state variable and one for $\mathbf{t}$. During simulation, the processor assigned *loc* sends changes to the technician's location to N other processors. Each processor assigned to a machine sends a message to the processor assigned to *loc* whenever the machine changes state. Finally, the processor assigned to $\mathbf{t}$ broadcasts changes to $\mathbf{t}$. □

Other mappings are possible. For example, consider relaxing the constraint in the above mapping that each state variable $S \in \mathcal{S}$ is represented by exactly one vertex in V_A. An alternative mapping assigns, for each value of m, couplings $c_{1,m}^{state}$ to a unique vertex in V_A, and assigns all remaining couplings to a single additional vertex in V_A, for a total of $N+1$ vertices in V_A. The result is a solution in which each machine effectively informs the technician when it fails, and then waits for the technician to respond when the machine has been repaired.

8 Mapping UNITY Simulation Programs to Simulation Protocols

We exemplify mapping the simulation program resulting from Section 7 to the time warp optimistic protocol [19]. Similar mappings can be devised for other parallel simulation protocols.

Time warp requires the time-of-next-event time flow mechanism (Section 6.2). 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 6.2, 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 straightforward, and hence is not illustrated.

9 UNITY-Based Methodology

We next propose a simulation program development methodology using the mechanism of the preceding sections. To state the methodology proposed precisely, we describe it in terms of Balci and Nance's simulation life cycle [2]. Assume that the "system and objectives definition" and "conceptual model" in the Balci and Nance life cycle have been completed [2]. 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 5) and stating rules to generate a UNITY program (cf. Section 4). 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: (Illustrated in Example 1) Define a *state variable* corresponding to each simulation model attribute. Enumerate all possible values of each state variable, and describe the constraints on transitions that the system can make between these values. State the initial value of each state 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: (Illustrated in Example 4) Formalize the CSTS of Step 1 in UNITY, using the rules stated in Section 5. The only verification required is to insure that the UNITY assertions have been correctly generated.

Overall verification of Step 1 to Step 2: (Illustrated in Example 6, below.) 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 3: (Illustrated in Example 2.) Derive a UNITY simulation program from the specification in Step 2 using the rules in Section 4. The only verification required is to ensure that the rules have been properly applied.

Step 4: Refine the simulation program by mapping the program to a particular time flow mechanism as described in Section 6.

Step 5: (Illustrated in Example 5) Refine the simulation program by mapping the program resulting from Step 4 to a particular sequential or parallel computer architecture as described in Section 7.

Step 6: Refine the simulation program by mapping the program resulting from Step 5 to a particular sequential or parallel simulation protocol as described in Section 8.

Example 6 The specification of Figure 11 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 Figure 11 and motivated our development of the CSTS semantics in Section 5.

P1 : $m.a \mapsto m.l$
P2 : $m.d \mapsto m.u$

Figure 13: Properties of MRP Used to Formally Verify Specification Correctness

We give two properties (Figure 13) which are proved in Appendix A. First, when the technician is at a particular machine, he eventually leaves that machine (P1). Second, when a machine goes down, it eventually comes back up (P2). Note that it is not possible to prove that an up machine eventually fails ($m.u \mapsto m.d$) from our specification; this requires reasoning about time. □

10 Conclusions

This paper presents a methodology to automate the construction of a simulation program from a communicative model. The model is specified as a 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 methodology applies to any simulation model that can be expressed using state variables whose domain is a finite set and which allows simultaneous events to be simulated in a non-deterministic order.

The proposed methodology may be incorporated into a simulation support environment which uses a higher level specification, such as an object oriented specification, than the CSTS by mapping that specification to a CSTS. The nature of these higher level specification forms is an given a specification describing only *single step transitions*, as Figure 11 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 to construct the formal chain of deductions required for each proof[7, 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.

To resolve issues in proofs versus graph-based analysis. Automated assistance in the verification and validation of simulation models through graph-based analysis techniques has been demonstrated in [25, 26]. Can these, or similar, techniques be applied to CSTS's? Can we define the relationship between what can be proved and what can be derived graphically?

To develop a variety of efficient mappings to parallel architectures and simulation protocols. Sections 7 and 8 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 open problem.

Other open problems which remain are the following:

To formally specify and verify properties about simulation time. The CSTS used in this paper can be mapped to the timed state transition system used by Henzinger, Manna, and Pnueli [16] for proofs of real time properties.

To formally specify and verify output measures. Output measures are often specified in terms of the time in which a predicate holds for a simulation model. Therefore UNITY is insufficient to specify output measures, and a specification language formalizing simulation time is required.

To formally specify and reason about ordering of simultaneous events. In the proposed methodology, simultaneous events are executed in a non-deterministic order.

To 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 (e.g., P1 and P2) must, at present, be done by hand. Automation is difficult because proofs of properties always requires identifying an order of application for UNITY theorems and sometimes requires formulation of invariants as well as metrics for induction, as illustrated in Appendix A. However, once generated, a proof can be checked automatically using Goldschlag's system [13].

Our experience in proving the properties of Figure 13 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 (P2) 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, 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.

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] Caine, S.H., and Gorden, E.K. (1975). "PDL – A Tool for Software Design," *Proceedings AFIPS NCC*, **44**, 271-276.

[6] Cameron, J.R. (1986). "An Overview of JSD," *IEEE Trans. on Software Eng.*, **SE-12** (2), February, 222-240.

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

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

[9] 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.

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

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

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

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

[14] Hamilton, M., and Zeldin, S. (1976). "Higher Order Software – A Methodology for Defining Software," *IEEE Trans. on Software Eng.*, **SE-2** (1), January, 173-190.

[15] Heacox, H.C. (1979). "RDL – A Language for Software Development," *ACM SIGPLAN Notices*, **14**, December, 71-79.

[16] 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

[17] Holbaek-Hanssen, E., Handlykken, P., and Nygaard, K. (1977). "System description and the DELTA Language," Report No. 4, Norwegian Computing Center, Oslo.

[18] Jackson, M.A. (1983). *System Development*, Prentice Hall, Englewood Cliffs, NJ.

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

[20] McArthur, D., Klahr, P., and Narain, S. (1984). *ROSS: An Object-Oriented Language for Constructing Simulations*, Rand Report R-3160-AF, The Rand Corporation, Santa Monica, CA, December.

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

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

[23] 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.

[24] Ören, T.I. (1984). "GEST – A Modeling and Simulation Language Based on System Theoretic Concepts," *Simulation and Model-Based Methodologies: An Integrative View*, T.I. Ören, B.P. Zeigler, M.S. Elzas, (eds.), Springer Verlag, NY, 281-335.

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

[26] 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.

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

[28] 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.

[29] Sanden, B. (1989). "An Entity-Life Modeling Approach to the Design of Concurrent Software," *Comm. ACM,* **32** (3), March, 330-346.

[30] 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.

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

[32] Teichrow, D., and Hershey, E.A. III. (1977). "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. on Software Eng.,* **SE-3** (1), January, 41-48.

[33] Zave, P. (1984). "The Operational Versus the Conventional Approach to Software Development," *Comm. ACM,* **27** (2), February, 104-118.

[34] Zave, P., and Schell, W. (1986). "Salient Features of an Executable Specification Language and its Environment," *IEEE Trans. on Software Eng.,* **SE-12** (2), February, 312-325.

[35] Zave, P. (1991). "An Insider's Evaluation of PAISLey," *IEEE Trans. on Software Eng.,* **SE-17** (3), March, 212-225.

A Proof of Properties P1 and P2

The proofs of P1 to P2 require the following UNITY theorems. Each theorem is written in the form of $\frac{\text{hypothesis}}{\text{conclusion}}$. In T4, W represents any set.

$$\text{T1}: \frac{p \mapsto q \wedge q \mapsto q'}{p \mapsto q'}$$

$$\text{T2}: \frac{p \mapsto q,\ q' \textit{ unless } b}{p \wedge q' \mapsto (q \wedge q') \vee b}$$

$$\text{T3}\frac{p \textit{ unless } q,\ p' \textit{ unless } q'}{(p \wedge p') \textit{ unless } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

$$\text{T4}: \frac{\langle \forall m : m \in W :: p(m) \mapsto q(m)\rangle}{(\langle \exists m : m \in W :: p(m)\rangle \mapsto \langle \exists m : m \in W :: q(m)\rangle)}$$

$$\text{T5}: \frac{p \mapsto q \Rightarrow q'}{p \mapsto q'}$$

$$\text{T6}: \frac{p \textit{ ensures } q}{p \mapsto q}$$

Theorems T1 to T6 are the transitivity property for leads-to, the progress safety progress theorem, the general conjunction theorem for *unless*, the disjunction theorem, and an inference rule for leads-to [7, pp. 52, 65, 58, 64-65, 52] Theorem T5 is consequence weakening for leads-to, and is straightforward to derive from other theorems.

We also require the following induction rule in our proofs [7, p. 72]. The rule refers to a well-founded set, which is a set that is partially ordered and has a lower bound.

> Let W be a set well-founded under the relation $\prec$. Let ME be a function, also called a *metric*, from program states to W; we write simply ME, without its argument, to denote the function value when the program state is understood from the context.
>
> The hypothesis of the rule is that from any program state in which p holds, the program execution eventually reaches a state in which q holds, or it reaches a state in which p holds and the value of metric ME is lower. Since the metric value cannot decrease indefinitely (from the well-foundedness of W), eventually a state is reached in which q holds.

$$\text{T7}: \frac{\langle \forall m : m \in W :: p \wedge (ME = m) \mapsto (p \wedge ME \prec m) \vee q \rangle}{p \mapsto q}$$

We will also require UNITY's *substitution axiom*: an invariant may always replace *true*.[7, p. 49]

A.1 Proof of Property P1

Proofs are written as a sequence of "deduction, justification" pairs.

We begin by proving two lemmas that will be used in the proofs of P1 and P2. First, if a machine is down and the technician is present, eventually the machine is in repair, and the technician is (still) present. Second, if the technician is at a machine in repair, then eventually the machine is up and the technician is present.

Lemma 2 $m.a \wedge m.d \mapsto m.a \wedge m.i$

Proof:

$m.a \wedge m.d$ *unless* $m.a \wedge m.i$

, Apply general conjunction for *unless* (T3) to MRP6 and MRP1

$m.a \wedge m.d \mapsto m.a \wedge m.i$

, Apply progress–safety–progress (T2) to last deduction and MRP6

□

Lemma 3 $m.a \wedge m.i \mapsto m.a \wedge m.u$

Proof: Apply progress–safety–progress (T2) to MRP7 and MRP1. $\square$

Proof of P1: $m.a \mapsto m.l$

$m.a \wedge m.i \mapsto m.l$

, Apply transitivity (T1) to Lemma 3 and MRP2.

$m.a \wedge m.d \mapsto m.l$

, Apply transitivity (T1) to Lemma 2 and previous deduction

$m.a \mapsto m.l$

, Apply disjunction (T4) to previous three deductions $\square$

A.2 Proof of Property P2:

First, we prove a lemma that will be used in the proof of P2: If a machine is down and the technician is not present, eventually the technician is present. The quantity $|m \ominus loc|$ used in the proof is a measure of the distance from the technician's current location to machine m.

Lemma 4 $m.d \wedge \neg m.a \mapsto m.d \wedge m.a$

Proof:

$m.l \mapsto (m \oplus 1).a$

, MRP3

$\langle \forall n : n \in \{0.0, 0.5, 1.0, 1.5, \ldots, N - 1.0, N - 0.5\} :: loc = n \mapsto loc = n \oplus 0.5\rangle$

, Combine last deduction with P1

$\langle \forall n : n \in \{0.0, 0.5, 1.0, 1.5, \ldots, N - 1.0, N - 0.5\} :: m.d \wedge \neg m.a \wedge loc = n \mapsto (m.d \wedge \neg m.a \wedge loc = n \oplus 0.5) \vee (m.d \wedge m.a)$

, Apply progress-safety-progress (T2) to last deduction and MRP5

$\langle \forall n : n \in \{0.0, 0.5, 1.0, 1.5, \ldots, N - 1.0, N - 0.5\} :: m.d \wedge \neg m.a \wedge |m \ominus loc| = n \mapsto (m.d \wedge \neg m.a \wedge |m \ominus loc| < n \oplus 0.5) \vee (m.d \wedge m.a)$

, Algebraic manipulation of last deduction

$m.d \wedge \neg m.a \mapsto m.d \wedge m.a$

, Apply induction (T7) to last deduction

□

Proof of Property P2: $m.d \mapsto m.u$

$m.d \wedge \neg m.a \mapsto m.i \wedge m.a$

, Apply transitivity (T1) to Lemmas 4 and 2

$m.d \mapsto m.i \wedge m.a$

, Apply disjunction (T4) to last deduction and Lemma 2

$m.i \wedge m.a \mapsto m.u$

, Apply consequence weakening (T5) to Lemma 3

$m.d \mapsto m.u$

, Apply transitive (T1) to last two deductions

□

B Proof of Conjectures

The proofs of the unless and leads-to conjectures for CSTS G requires the code that is equivalent to G, but with all time formulas equal to zero (See Lemma 1.). The code is shown in Figure 14.

B.1 Proof of *unless* Conjectures

We prove the second conjecture in Figure 9; the remaining *unless* conjecture proofs are similar.

Proof of $m.a \wedge m.u \overset{?}{\textit{unless}} m.l$**:** Using the definition of *unless*, the verification condition is:

$$\langle \forall \text{ statements } s \text{ in MRP} :: \{m.a \wedge m.u\}s\{(m.a \wedge m.u) \vee m.l\}\rangle$$

The proof consists of instantiating the condition above with each value of s to show that s either performs an identity assignment or establishes $m.a$. For the statement s corresponding to $c_{1,m}^{loc}$, $\{m.a \wedge m.u\}s\{m.l\}$. Execution of all other statements s results in the identity assignment; therefore $\{m.a \wedge m.u\}s\{m.a \wedge m.u\}$. Hence the conjecture holds.

□

```
program MRP
  constants N=...; MaxRepairs=...; T=...; λ[N]=...; μ[N]=...;
  declare
    loc        : (1,1.5,...,N,N+0.5)
    state[N]   : (up, inrepair, down)

  initially
    || ⟨ || m :: state[m]=up ⟩        { initially all machines are up }
    || loc=0.5                        { technician leaving machine zero}

  assign

  {Implementation of $c^{loc}_{1,m}$:}
    □⟨ □ m :: loc := loc ⊕ 0.5            if loc=m ∧ state[m]=up ⟩

  {Implementation of $c^{loc}_{2,m}$:}
    □⟨ □ m :: loc := loc ⊕ 0.5            if loc=m⊕0.5 ⟩

  {Implementation of $c^{state}_{1,m}$:}
    □⟨ □ m :: state[m] := down            if state[m]=up ∧ loc ≠ m ⟩

  {Implementation of $c^{state}_{2,m}$:}
    □⟨ □ m :: state[m] := inrepair        if state[m]=down ∧ loc=m ⟩

  {Implementation of $c^{state}_{1,m}$:}
    □⟨ □ m :: state[m] := up              if state[m]=inrepair ⟩

end { MRP }
```

Figure 14: UNITY Program for Machine Repairman Problem with time formulas equal to zero.

B.2 Proof of *leads-to* Conjectures

We prove the first conjecture in Figure 10; the remaining *unless* conjecture proofs are similar. (The one conjecture which cannot be proven, $m.u \land \neg m.a \overset{?}{\mapsto} m.d$ actually holds for the CSTS, but not for the zero time code. The conjecture cannot be proven without a proof system that can reason about simulation time, because the proof depends on the fact that a machine goes down after a finite time elapses.)

Proof of $m.a \land m.u \overset{?}{\mapsto} m.l$: From Section B.1, $m.a \land m.u$ *unless* $m.l$ holds. For the statement s in program MPR corresponding to $c^{loc}_{1,m}$, $\{m.a \land m.u\}s\{m.l\}$. Hence $m.a \land m.u$ *ensures* $m.l$. By T6, the conjecture holds.

□