Linking Simulation Model Specification and Parallel Execution Through UNITY

By Marc Abrams, Ernest Page,
and Richard E. Nance

TR 91-14

Linking Simulation Model Specification and Parallel Execution through UNITY

Marc Abrams
Ernie Page
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24060-0106

Richard E. Nance
Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, VA 24060-0251

TR 91-14

May 10, 1991

Abstract

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

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, UNITY, parallel simulation protocols, simulation verification

Contents

1 Introduction

The automated support of simulation model development is entering the second decade as a topic of significant research interest. Approaches to computer assistance have sought a conceptual basis in artificial intelligence [8, 12], general systems theory [7, 9], software engineering [6], and modeling methodologies [2, 11]. In fact, the primary efforts in simulation support environments draw to varying degrees from all these conceptual sources. Balzer, Cheatham, and Green [3, p. 41] describe the automation-based paradigm as separating implementation from specifications so that maintenance is performed entirely on the latter. Automatic translation from a higher level specification to an efficient implementation is envisioned. This perspective on application development and support emphasizes the role of specification languages (see [13] for an excellent survey) and the necessity for realizing an efficient implementation.

Simulation modeling represents a challenge for both model specification and implementation, and this work represents an effort to assess the potential of UNITY [4] for accomplishing both. In addition, UNITY is intended for development of efficient parallel and distributed programs through step-wise refinement of specifications. This paper also assesses the potential of UNITY to derive efficient parallel simulation implementations.

2 Program Specification in UNITY

The goal of UNITY is to provide a means to systematically develop programs for a wide variety of applications and computer architectures. Architectures considered include sequential, synchronous and asynchronous shared-memory multiprocessor, and message-based distributed processor.

2.1 Program Development by Stepwise Refinement

UNITY supports program development as a 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. In this way, 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 that at each stage of refinement, one must prove that the refined specification implies a specification

proposed at an earlier step. In addition, one must prove that the program derived from the most refined specification meets that specification.

The main contribution of UNITY is a computation model that underlies a wide variety of computer architectures, and a proof system. In addition, the proof system allows proof of both *safety* and *progress* properties. A safety property of a program holds in all computation states, such as an invariant. A progress property is a property that holds in a particular program state. An example of a safety property is that if i is a program variable and A is an array of N elements, then at any point during execution of the program, elements $A[1], \ldots, A[i]$ are sorted. An example of a progress property is that eventually all elements of the array A will be sorted (e.g., every execution of the program must reach a computation state in which $i = N$). Historically, progress properties have been much more difficult to prove than safety properties; UNITY is comprehensive in its ability to prove both types of properties with one proof system.

2.2 UNITY Computation Model and Proof System

"A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple assignment statements" [4]. The UNITY computation 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" [4]. Note that the computation model represents simultaneous execution of assignments in a parallel computer by interleaved execution.

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

2.3 Absence of Control Flow

Control flow constrains the order in which assignment statements are executed. Examples of control flow in imperative programming languages, such as FORTRAN or C, include if statements, do and while loops, and subroutine calls. UNITY is founded on the belief that one of the things that makes writing parallel programs hard is that we are accustomed to over-specifying control flow from our experience with sequential programs. In fact, efforts to automatically transform sequential FORTRAN programs to parallel programs require code analysis to identify what control flow constraints can be relaxed.

The UNITY goal of postponing questions of efficiency and architecture to later in the refinement process is made possible by saying very little about

the *order* in which assignments are executed at early specification stages, and by including with a program control flow in the form of a detailed execution schedule of assignments that is efficient for a target architecture.

2.4 UNITY Programming Logic

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.

UNITY defines three fundamental logical relations: *unless*, *ensures*, and *leads-to*. The definitions below are those of Chandy and Misra [4, Chapter 3].

Unless: For a given program F, "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,

$$p \text{ unless } q \equiv \langle \forall s : s \text{ in } F :: \{p \wedge \neg q\}\, s\, \{p \vee q\} \rangle.$$

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,

$$p \text{ ensures } q \equiv p \text{ unless } q \wedge \langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\, s\, \{q\} \rangle.$$

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. The paper requires 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,

$$p \text{ until } q \equiv (p \text{ unless } q) \wedge (p \mapsto q).$$

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

Figure 1: A UNITY program to sort array A into ascending order.

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 $false$

2.5 Example: Sorting

As an example, consider the following problem [4]: Sort integer array $A[0..N]$, $N \geq 0$, in ascending order. The specification of the sort program says firstly 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 $i = 0, 1, \ldots, N$. 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. The symbol "$\Box$" in the program acts as a separator between assignment statements. UNITY generally uses Pascal syntax for variable declarations, assignment statements, and expressions. The quantification over variable i in the problem along with the $\Box$ denotes N instances of the assignment statement, equivalent to:

$$\begin{array}{ll} A[0], A[1] := A[1], A[0] & \text{if } A[0] > A[1] \\ A[1], A[2] := A[2], A[1] & \text{if } A[1] > A[2] \\ \ldots & \\ A[N-1], A[N] := A[N], A[N-1] & \text{if } A[N-1] > A[N]. \end{array}$$

2.6 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.

The union of two programs results from appending the code of both programs together. The union rule will not be used in the example problem of this paper, and hence will not be discussed further, however the union rule is useful to simulation modelers.

The superposition rule will be used later in this paper. 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" [4].

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

1. 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.

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

Ideally we would like to be able to refine a simulation model specification into a simulation program, and then refine the simulation program so that it contains an efficient time flow mechanism and can be efficiently mapped to a target architecture. The approach used in Section 4.4 is to layer the simulation model on the time flow mechanism, which in turn is layered on the parallel simulation protocol. Superposition will be used to specify this layering.

2.7 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, item 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" [4].

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 computation 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. In fact, UNITY can model refinement down to the level of electronic circuits.

3 Simulation Specification in UNITY

We propose that a simulation model be represented as a UNITY program by mapping simulation "attributes" and "events," as defined by Kiviat [11], to UNITY "variables" and "assignment statements," respectively.

Assume that the "system and objectives definition" and "conceptual model" in Balci and Nance's simulation life cycle are completed [1]. We propose using a state transition diagram representation of the "communicative model" in the methodology to simplify the presentation. Starting at this point we propose the following methodology:

Step 1: This step specifies a simulation that captures the *order* of events that occur in the system, but ignores the absolute *time* at which events occur. (For the machine repairman problem discussed in Section 4, this means capturing the correct state space and state transitions without regard to failure and repair rates or the rate at which the operator walks.)

- *Step 1A:* Select a set of *states* and formulate a state transition diagram. Verify that the state transition diagram and the conceptual model match. (It is possible to use multiple state transition diagrams, but this is beyond the scope of the paper.)
- *Step 1B:* Express each output measure in terms of the holding time for a set of states. Verify that all output measures can be expressed in terms of the states selected in Step 1A.
- *Step 1C:* Formalize the state transition diagram of Step 1A in UNITY. Verify that all transitions present (prohibited) in the diagram match transitions present (respectively, prohibited) in the UNITY specification.
- *Step 1D:* Express any additional properties that the composite model (i.e., the union of the modules) must satisfy. The specification consists of these additional properties along with the UNITY assertions of

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

Step 2: Refine the simulation by mapping the order of events to a time scale. (In the machine repairman problem of Section 4, this means adding failure and repair rates and the rate at which the operator walks.) Verify that the refined specification meets the specification from Step 1.

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

Step 4: Refine the simulation program by mapping the program to a particular

1. time flow mechanism,
2. sequential or parallel simulation protocol, and
3. sequential or parallel hardware architecture.

We conjecture that these three must all be considered together to achieve an efficient program.

4 The Machine Repairman Problem

This paper applies the methodology of Section 3 to the classical machine interference problem [5]. In the problem, a set of N semi-automatic machines fail intermittently and are repaired by one or more technicians. Machine failure rates are assumed to follow a Poisson distribution with parameter λ. Upon arriving at a failed machine, a technician can repair the machine in a time period that is exponentially distributed with parameter μ. A variety of service disciplines are possible that specify how the technician selects a machine to repair.

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

This paper considers the patrolling repairman service discipline, in which a ***single*** technician services all machines [10, p. 60]. The technician traverses a path amongst the machines in a cyclic fashion $(1, 2, \ldots, N, 1, \ldots)$. The rate at which the technician walks is a constant, T. This problem, hereafter referred to

as the *machine repairman problem* (MRP), is chosen so that both the UNITY specification and program may be presented within the space available for this paper.

4.1 Illustration of Methodology Step 1

4.1.1 Step 1A: Select State Transition Diagram

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

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

$$m.u \equiv (m.state{=}up)$$

$$m.d \equiv (m.state{=}down)$$

$$m.i \equiv (m.state{=}inrepair)$$

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

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

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

$$m.a \equiv (location{=}\ m)$$

$$m.l \equiv (location{=}\ m + 0.5)$$

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

State Transition Diagram: The state of the system is represented by $N+1$ state variables: $\forall m,\ m = 1, 2, \ldots, N$, $m.state$ and *location*. Figure 2 illustrates the state transition diagram by illustrating just two state variables: $m.state$ and *location*. All possible transitions are illustrated in this diagram. For example, if in the current system state machine m is *up* and the technician is at machine

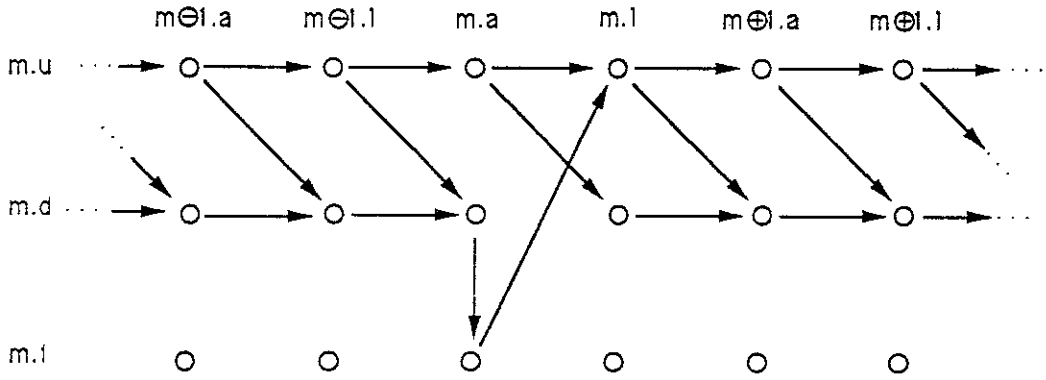

Figure 2: Portion of state transition diagram illustrating variables *m.state* and *location*.

m, after the next state change the technician must be leaving machine m and machine m will either still be up or it will be down. However it cannot be the case, if in the current system state machine m is *up* and the technician is not at machine m, that after the next state change machine m is in repair, or the technician is still at machine m. Also note from the diagram that the only way that a machine can enter the *in repair* state is if in the current state the technician is at that machine and the machine is down. If a machine is in repair, after the next state the machine state must be *up* and the technician must be leaving that machine.

4.1.2 Step 1B: Express Output Measures Using Holding Times

Let us assume that the desired output measures are:

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

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

$$traveling \equiv (\forall m :: m.l)$$

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

4.1.3 Step 1C: Formalize State Transition Diagrams in UNITY

The UNITY specification of the MRP is shown in Figure 3. In all UNITY formulas in the paper, universal quantification over the values of variable *location* is assumed, unless the quantification is explicit. Hence formulas MRP1 to MRP5, MRP7, and MRP9 hold for $m = 1, 2, \ldots$.

Formulas MRP1 through MRP5 represent the state transition diagram of Figure 2, and will be discussed next. Let $a \oplus b \equiv (a + b) \bmod N + 0.5$ and $a \ominus b \equiv (a - b) \bmod N + 0.5$. First, if in the current system state a machine is up and the technician is at location n, the system will remain in this state until, eventually, the system changes state; in the new state the technician will have advanced to the next location ($N \oplus 0.5$) and the machine will either still be up or will have gone down (MRP1). Second, if in the current state a machine is down, the machine remains down until, eventually, a technician arrives at the machine and the machine is in repair (MRP2). Third, the technician remains at a machine in repair until the machine goes up and the technician leaves the machine (MRP3). Fourth, a machine which is down remains down while the technician is not at the machine; furthermore the technician continues to advance from one location to the next location (MRP4). Fifth, the system is never in a state in which a machine is in repair without the technician present (MRP5).

4.1.4 Step 1D: Express Additional Properties

Four properties not reflected in the state transition diagram (Figure 2) must be added. In the first three properties, integer variable NR denotes the number of times the technician has completed a repair. Initially, $NR = 0$ (MRP6). If the technician is at a machine that is in repair, the value of NR will increase by one when the technician leaves the machine or the machine is no longer in repair (MRP7). If the technician is not at a machine in repair, then the value of NR does not change (MRP8). The final property guarantees that a machine is not up forever (MRP9).

Verification: Step 1D 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 increases our confidence in the specification. In fact, in writing this paper our original statement of the specification omitted several properties shown in Figure 3.

MRP1 : $m.u \wedge location = n$ ensures $(m.u \vee m.d) \wedge location = n \oplus 0.5$

MRP2 : $m.d \wedge m.a$ ensures $m.i \wedge m.a$

MRP3 : $m.i \wedge m.a$ ensures $m.u \wedge m.l$

MRP4 : $m.d \wedge \neg m.a \wedge location = n$ ensures $m.d \wedge location = n \oplus 0.5$

MRP5 : $\neg(m.i \wedge \neg m.a)$ is invariant

MRP6 : initial condition $\Rightarrow NR = 0$

MRP7 : $NR = k \wedge m.a \wedge m.i$ ensures $NR = k + 1 \wedge \neg(m.a \wedge m.i)$

MRP8 : $NR = k \wedge \langle \forall m :: \neg(m.a \wedge m.i) \rangle$ unless
$NR = k \wedge \langle \exists m :: (m.a \wedge m.i) \rangle$

MRP9 : $m.u$ ensures $\neg m.u$

Figure 3: UNITY specification for the MRP

We give six properties (Figure 4), whose proofs appear in the Appendix. First, when a machine goes down, it is eventually repaired and comes back up (P1). Second, a machine which is up remains up, unless it goes down (P2). (This prohibits a transition from *up* to *in repair*.) Third, when the technician is at a particular machine, he remains at that machine until, eventually, he leaves that machine (P3). Fourth, if the technician is leaving machine m, he eventually changes location to being at machine $m \oplus 1$ (P4). Fifth, at most one machine can be in repair at any time (P5). Finally, the value of variable NR eventually exceeds any constant *MaxRepairs* (P6).

4.2 Illustration of Methodology Step 2

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

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

The specification will represent "calls to a random number generator" by referring to a sequence whose elements are a list of random variates returned

P1 : $m.d \mapsto m.u$

P2 : $m.u$ unless $m.d$

P3 : $m.a$ until $m.l$

P4 : $m.l$ until $(m \oplus 1).a$

P5 : $\forall m, n \; : \; m \neq n \; :: \; \neg(m.i \wedge n.i)$

P6 : $true \mapsto NR \geq MaxRepairs$

Figure 4: Properties of the MRP used to formally verify the correctness of the specification.

by the random number generator. Let sequence $m.\lambda$ denote a list of random variates representing the sequence of times for which machine m remains up. Let sequence $m.\mu$ denote a list of random variates representing the sequence of repair times of machine m.

The additional assertions are:[1]

1. The holding time of state $m.u$ is Head(m.λ).
2. The holding time of state $m.a \wedge m.i$ is Head(m.μ).

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

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

4.3 Illustration of Methodology Step 3

The specification of Figure 3 is implemented by program MRP, shown in Figures 5 to 6.

Verification: Formal proof that the code meets the specification in Figure 3 can be carried out, but is not presented in this paper. Proof that the output measures are correctly computed requires formulating and proving a suitably strong invariant.

[1] Missing from the assertions is an explanation of when each sequence has its head removed.

program *MRP* {simulate the MRP}

declare

constants

N=...	{positive integer}
MaxRepairs=...	{nonnegative integer}
T=...	{positive integer, denoting travel time}

types

alarm = integer

variables

m	: integer	{machine number; integer in [1,N]}
State[N]	: (up, down, inrepair)	{enumerated type}
Location	: (1,1.5,...,N,N+0.5)	{enumerated type}
NR	: integer	{number of completed repairs}
SysTime	: integer	{current simulation time; read only }
Failure[N]	: alarm	{Failure[m] is time of next failure of machine m if Failure[m]>SysTime }
Arrival[N]	: alarm	{Arrival[m] is time of next technician arrival at machine m, if m⊖0.5=Location, otherwise Arrival[m] is last time at which technician arrived at machine m}
Finish[N]	: alarm	{Finish[m] is time of next repair completion of machine m if Location=m ∧ State[m]=down, otherwise Finish[m] is time of previous repair completion of machine m}
λ[N]	: sequence of integer	{λ[m] is sequence of random variates representing time between failures; initialized outside of MRP}
μ[N]	: sequence of integer	{μ[m] is sequence of random variates representing repair times; initialized outside of MRP}

always

term = NR $\geq$ MaxRepairs

Figure 5: UNITY code for MRP (continued in next figure)

initially
SysTime = 0.0 || NR = 0
|| Location=1.5 { technician initially leaving machine 1}
|| ⟨ || m : $1 \leq m \leq N$ :: State[m] = up ⟩ {initially all machines are up }
|| ⟨ || m : $1 \leq m \leq N$ :: Failure[m] = SysTime + Head(λ[m])
|| λ[m] = Tail(λ[m])
⟩
|| ⟨ || m : $1 \leq m \leq N$ :: Arrival[m] = SysTime + T
if m = Location $\oplus$ 0.5
|| Arrival[m] = -∞ if m $\neq$ Location $\oplus$ 0.5
⟩

assign
{ The assignment section basically deals with 2 events: arrival
and finish (repair). }

{ Arrival: Update location, set machine state to inrepair,
and schedule finish if machine is down, update location (again)
and schedule arrival at next machine otherwise. }
□ ⟨ || m : $1 \leq m \leq N$ :: Location[m] := m
if SysTime = Arrival[m] $\wedge$ $\neg$term
□ State[m], Finish[m], μ[m] := inrepair, SysTime+Head(μ[m]), tail(μ[m])
if Location=m $\wedge$ State[m]=down $\wedge$ $\neg$term
□ Location, Arrival[(m $\oplus$ 1)] := m $\oplus$ 0.5, SysTime+T
if Location=m $\wedge$ State[m]=up $\wedge$ $\neg$term
⟩

{ Finish: Increment NR, set machine state to up, schedule next
failure, update technician's location and schedule arrival at
next machine}
□ ⟨ || m : $1 \leq m \leq N$ :: NR, Arrive[m $\oplus$1], Failure[m], State[m], Location, λ[m] :=
NR+1, SysTime+T, SysTime+Head(λ[m]), up, m $\oplus$0.5, Tail(λ[m])
if SysTime=Finish[m] $\wedge$ $\neg$term
⟩

{ Failure: Set machine's state to down. }
⟨ || m : $1 \leq m \leq N$ :: State[m] := down
if SysTime=Failure[m] $\wedge$ $\neg$term
⟩
end { *MRP* }

Figure 6: Continuation of UNITY code for MRP

It is impossible to prove the time-in-state assertions from Section 4.3 using the current proof system of UNITY. UNITY's computation model of fairly interleaved, atomic execution of statements permits no notion of simultaneity, which means that deep changes to UNITY are required to carry out these proofs.

4.4 Illustration of Methodology Step 4

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

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

4.4.1 Superposing Fixed Time Increment

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

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

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

2. Set *SysTime*to *SysTime*+ Δ.

In order to add the above two phase algorithm to the Step 3 specification (Figures 5 and 6) we must devise a means to insure that *all* statements whose alarms have gone off at the current value of *SysTime* are executed *before SysTime* is incremented. (Because UNITY does not specify sequencing of statements, we must add something to enforce the two phase sequencing.)

Let the UNITY program of Step 3 contain S statements in the **assign** section ($S = 5$ in Figures 5 and 6). To enforce the two phase algorithm, we first number

Program *FTI_TFM*
declare A[S] : integer
initially $\langle i : 1 \leq i \leq A[i] = 0 \rangle$
transform

each statement s in the underlying program to $s \parallel A[i] := 1$ where i is the lexical statement number of s.

add to always section

update = $\langle \wedge i : 1 \leq i \leq S :: A[i] = 1 \rangle$

add to assign section

$\langle \parallel i :: 1 \leq i \leq S :: A[i] := 0 \quad$ if update $\rangle \parallel SysTime := SysTime +$
$\Delta \quad$ if update

end { *FTI_TFM* }

Figure 7: Specification of fixed time increment time flow mechanism

the statements in the program of Step 3 by the integers 1,2,...,S. Next we add array $A[1..S]$. Initially, all elements of array A are zero. Each statement s_i numbered i (for $1 \leq i \leq S$) is transformed to $s_i \parallel A[i] := 1$. When all elements of array A are one, *SysTime* can be incremented. When system time is incremented (in the superposed program) all elements of array A are set to zero.[2]

The superposed program is formalized in Figure 7. Note that Figure 7 works with *any* simulation specification that results from Step 3.

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

4.4.2 Superposing Time-of-Next-Event

Next we sketch a method to add the next event time flow mechanism to a Step 3 program. As in the fixed time increment method, we assign each statement in the assign section an integer identification number. These numbers serve as event numbers. We add an EventList and a variable called CurrentEvent. Recall that m is an integer in $[1, \ldots, N]$ denoting a machine number. EventList is a list of triples (time, event number, m). The statements which set alarms in Figures 5 and 6 now append triples to EventList. The time flow mechanism

[2]Note that an assignment statement of the form $x := e$ if b in Figures 5 and 6 is a shorthand for $x := e$ if $b \| x := x$ if $\neg b$. Therefore the statement is executed even though b is false.

superimposed on the program will set *SysTime* to the time component of a triple of EventList that is less than or equal to the time component of all other triples. This triple's event number is stored in CurrentEvent. Finally, We add to each statement s_i in the assign section the condition "if CurrentEvent=s_i."

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

4.4.3 Mapping Specification to a Protocol and Architecture

Mapping a simulation specification to a simulation protocol is an open problem. Mapping of UNITY specifications to architectures is discussed by Chandy and Misra [4, Ch. 4], and applies to simulation specifications.

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

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

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

5 Conclusions

Step 1 of the proposed methodology dictates that the *order* of events in a conceptual model be correctly specified without regard for the particular times at which events occur. The justification is that one often wishes to "get the simulation logic correct." Based on the example in Section 4.1, UNITY works well for this job.

Step 2 (mapping the order of events to a time scale) requires a modification of UNITY to add notation for the holding time of certain states. We introduced such a notation in Section 4.2. However, in order to prove any properties about timings, the UNITY proof system must be extended, which is likely to be a difficult task.

Step 3 (deriving a simulation program from the specification) in Section 4.3 is straightforward. Again, we cannot formally verify the correctness of the timing properties without an extension of the proof system to handle time.

Step 4 (mapping the program to a particular time flow mechanism, sequential or parallel simulation protocol, and sequential or parallel target architecture) requires additional research to accomplish. Based on the example in Section 4.4 of mapping the MRP to fixed time increment as well as the Time-of-Next-Event mechanisms, we believe UNITY is sufficient to handle Step 4.

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

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

(a) **Applying induction:** Consider the proof that down machines are eventually repaired (P1) in the Appendix. A key to the proof is establishing by an induction proof (using theorem T11) that after a machine goes down, the technician keeps getting "closer" to the failed machine, until eventually he is at the failed machine. Induction is required whenever we want to draw a conclusion about a *sequence* of state transitions, given a specification describing only *single step transitions*, as Figure 3 does. Figuring out how to fit the induction theorem (T11) 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 [4, Ch. 3] to construct the formal chain of deductions required for each proof. This process is somewhat analogous to what an undergraduate student does in a calculus class, as he browses through a table of integrals and a list of trig identities in trying to symbolically integrate a function. However a theorem proving system could alleviate this problem.

(c) **Devising invariants:** This paper did not present a proof that the simulation code (Figures 5 and 6) meets the specification. However, proofs of code generally require invariants to be formulated, which requires 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 decline.

Automation-based paradigm: If UNITY grows in popularity, a rich set of methods to map UNITY programs to target architectures may be developed. By identifying the correspondence between simulation modeling and UNITY programs, a model development environment using the automation-based paradigm could apply the UNITY architecture mappings for simulation models to assist in construction of parallel simulation programs.

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

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

References

[1] O. Balci (1989), "How to Assess the Acceptability and Credibility of Simulation Results," In *Proceedings of the 1989 Winter Simulation Conference*, E. A. MacNair, K. J. Musselman, and P. Heidelberger (eds.), pp. 62-71.

[2] Balmer, D.W. and R.J. Paul (1986), "CASM – The Right Environment for Simulation," *Journal of the Operational Research Society 37*, pp. 443-452.

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

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

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

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

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

[8] Klahr, P. (1985), "Expressibility in ROSS: An Object-Oriented Simulation System," In: *AI Applied to Simulation: Proceedings of the European Conference at the University of Ghent*, pp. 136-139.

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

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

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

[12] Snyder, J. and G.T. Macbulack (1988), "Intelligent Simulation Environments: Identification of the Basics," In: *Proceedings of the 1988 Winter Simulation Conference*, M. Abrams, P. Haigh, and J. Comfort (eds.), pp. 359-363.

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

A Proof of Properties P1 to P6

This appendix contains proofs of the properties of Figure 4, and requires the following UNITY theorems. In T9, W represents any set.

T1 : $(p \text{ unless } q \wedge q \text{ unless } r) \Rightarrow (p \vee q \text{ unless } r)$

T2 : $(p \text{ unless } q \wedge q \Rightarrow r) \Rightarrow (p \text{ unless } r)$

T3 : $(p \text{ unless } q \wedge p' \text{ unless } q') \Rightarrow (p \vee p' \text{ unless } q \vee q')$

T4 : $(p \text{ ensures } q \wedge q \Rightarrow r) \Rightarrow (p \text{ ensures } r)$

T5 : $(p \text{ ensures } q) \Rightarrow (p \mapsto q)$

T6 : $(p \mapsto q \wedge q \mapsto r) \Rightarrow (p \mapsto r)$

T7 : $(p \mapsto q \wedge p' \mapsto q) \Rightarrow (p \vee p' \mapsto q)$

T8 : $(p \mapsto q \wedge r \text{ unless } b) \Rightarrow (p \wedge r \mapsto (q \wedge r) \vee b)$

T9 : $\langle \forall m : m \in W :: p(m) \mapsto q(m) \rangle \Rightarrow$
$(\langle \exists m : m \in W :: p(m) \rangle \mapsto \langle \exists m : m \in W :: q(m) \rangle)$

T10 : $(p \mapsto q \Rightarrow r) \Rightarrow (p \mapsto r)$

The theorems expressed by T1 to T9 are found in the UNITY book on pages 59, 58, 59, 62, 52, 52, 52, 65, and 64-65 respectively [4]. It is straightforward to derive T10 from other UNITY theorems.

We will also require the following induction rule in our proofs [4, 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 M be a function, also called a *metric*, from program states to W; we write simply M, 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 M is lower. Since the metric value cannot decrease indefinitely (from the well-foundedness of W), eventually a state is reached in which q holds.

T11 : $(\langle j : j \in W :: p \wedge (M = j) \mapsto (p \wedge (M \prec j) \vee q) \rangle) \Rightarrow (p \mapsto q)$

We will also require UNITY's *substitution axiom*: *true* may always be replaced by an invariant (e.g., MRP5) [4, p. 49].

Finally, we prove a lemma that will be used in two of the proofs: If a machine is down and the technician is not present, eventually the technician is present. The proof list a sequence of "deduction, justification" pairs. The quantity $|m \ominus location|$ used in the proof is a measure of the distance from the technician's current location to machine m.

Lemma 1: $m.d \wedge \neg m.a \mapsto m.d \wedge m.a$

Proof:
$\langle \forall n : n \in \{1, 1.5, 2, 2.5, \ldots, N, N + 0.5\} :: m.d \wedge \neg m.a \wedge |m \ominus location| = n \mapsto$
$(m.d \wedge \neg m.a \wedge |m \ominus location| < n) \vee (m.d \wedge m.a) \rangle$
, MRP4
$m.d \wedge \neg m.a \mapsto m.d \wedge m.a$
, apply T11 to last deduction □

A.1 Proof of Property P1

The proof of property P1, that a down machine is eventually repaired, is given below.

$m.d \wedge m.a \mapsto m.u$
, MRP2 $\wedge$ MRP3 $\wedge$ T5 $\wedge$ T6 $\wedge$ T10
$m.d \wedge \neg m.a \mapsto m.u$
, apply T6 to last deduction and Lemma 1
$m.d \mapsto m.u$
, first and last deductions $\wedge$ T7 □

A.2 Proof of Property P2

$m.u$ ensures $(m.u \vee m.d)$
, $(\exists n :: location = n \Rightarrow true) \wedge$ T4
$m.u$ unless $m.u \vee m.d$
, definition of ensures
$m.u$ unless $m.d$
, definition of unless applied to last deduction □

A.3 Proof of Property P3

According to the definition of *until*, to prove P3, that the technician remains at a machine until he leaves that machine, requires establishing two properties: $m.a$ unless $m.l$ and $m.a \mapsto m.l$. These properties are established below in deductions (f) and (j), respectively.

a: $m.d \wedge m.a$ unless $m.i \wedge m.a$
, MRP2 $\wedge$ definition of ensures
b: $m.i \wedge m.a$ unless $m.u \wedge m.l$
, MRP3 $\wedge$ definition of ensures
c: $(m.d \vee m.i) \wedge m.a$ unless $m.u \wedge m.l$
, T1
d: $m.u \wedge m.a$ unless $(m.u \vee m.d) \wedge m.l$
, MRP1 $\wedge$ definition of ensures
e: $m.a \wedge (m.u \vee m.d \vee m.i)$ unless $m.l \wedge (m.u \vee m.d)$
, apply T3 to last two deductions
f: $m.a$ unless m.l
, apply T2 to last deduction $\wedge$ $(m.a \vee m.d \vee m.i \Rightarrow true)$
g: $m.i \wedge m.a \mapsto m.l$
, MRP3 $\wedge$ T5 $\wedge$ T4
h: $m.d \wedge m.a \mapsto m.l$
, MRP2 $\wedge$ T5 $\wedge$ (g) $\wedge$ T6
i: $m.u \wedge m.a \mapsto m.l$
, MRP1 $\wedge$ T5 $\wedge$ T4

j: $m.a \mapsto m.l$
, apply T7 twice to last three deductions $\wedge\ m.i \vee m.d \vee m.u \Rightarrow true$
k: $m.a$ until $m.l$
, definition of *until* applied to (f) and (j) □

A.4 Proof of Property P4

Analogous to the proof of P3, proof of P4 requires establishing $m.a$ unless $(m \ominus 1).l$ and $m.a \mapsto (m \oplus 1).l$, which is done below in deductions (c) and (e), respectively.

$a : (m.u \vee m.d) \wedge m.l$ unless $(m \oplus 1).a$
, MRP1 $\wedge$ MRP4 $\wedge$ definition of ensures $\wedge$ T3 $\wedge$ T2
$b : (\neg m.i \vee \neg m.l) \wedge (m.u \vee m.d) \wedge m.l$ unless $(m \oplus 1).a$
, substitute MRP5 into last deduction via substitution axiom
$c : m.l$ unless $(m \ominus 1).a$
, last deduction $\wedge\ (\neg m.i \wedge (m.u \vee m.d) \Rightarrow true\ )$
$d : (m.u \vee m.d) \wedge m.l \mapsto (m \oplus 1).a$
, MRP1 $\wedge$ MRP4 $\wedge$ T4 $\wedge$ T5 $\wedge$ T7
$e : m.l \mapsto (m \oplus 1).a$
, substitution axiom on MRP5, last deduction $\wedge\ (\neg m.i \wedge (m.u \vee m.d) \Rightarrow true\ )$
$f : m.d$ until $(m \ominus 1).a$
, definition of until applied to (c) and (e) □

A.5 Proof of Property P5

P5 will be proved by contradiction.

$m.i \wedge n.i$
, negation of the conclusion of P5
$m.i \Rightarrow m.a$
, MRP5 $\wedge\ (a \Rightarrow b \equiv b \vee \neg a)$
$m.a \wedge n.a$
, combine last two deductions
$false$
, $m.a \wedge n.a \Rightarrow false$
$\neg(m.i \wedge n.i)$
, last deduction implies conclusion of P5 holds □

A.6 Proof of Property P6

P6 states that in any execution sequence the value of variable NR will exceed any constant *MaxRepairs*. The specification, in MRP7 and MRP8, states that NR increases in value only when the technician leaves a machine that is in repair. Therefore we first show, in Lemma 2 below, that the system forever cycles back

and forth between a state in which the technician is present at a machine in repair and a state in which the technician is not present at a machine in repair. The proof of P6 uses the lemma to establish that NR forever increases in value, and then uses the induction theorem (T11) to show that NR continues to get "closer" in value to *MaxRepairs*until it eventually equals or exceeds *MaxRepairs*. The metric is $M = \max(0, \mathit{MaxRepairs} - NR)$.

Lemma 2: $\neg(m.a \wedge m.i) \mapsto m.a \wedge m.i$

Proof:

a: $\neg(m.a \wedge m.i) \equiv (m.a \wedge m.d) \vee (\neg m.a \wedge m.d) \vee m.u$
, $(\neg m.i \equiv m.u \vee m.d) \wedge$ MRP5

b: $m.a \wedge m.d \mapsto m.a \wedge m.i$
, MRP2 $\wedge$ T5

c: $\neg m.a \wedge m.d \mapsto m.a \wedge m.i$
, Combine result of applying T5 to MRP2 and Lemma 1 using T6

d: $m.u \mapsto m.d \vee m.i$
, MRP9 $\wedge$ T5

e: $m.u \mapsto m.d$
, T8 applied to last deduction $\wedge$ P2

f: $m.d \mapsto m.a \wedge m.i$
, apply T7 to (b) and (c)

g: $m.u \mapsto m.a \wedge m.i$
, apply T7 to last two deductions

h: $\neg(m.a \wedge m.i) \mapsto m.a \wedge m.i$
, combine (b), (c), and (g) using T7 twice and (a) □

Proof of P6:

$\langle \exists m :: \neg(m.a \wedge m.i) \rangle \mapsto \langle \exists m :: m.a \wedge m.i \rangle$
, T9 applied to Lemma 2

$NR = k \wedge \langle \forall m :: \neg(m.a \wedge m.i) \rangle \mapsto NR = k \wedge \langle \exists m :: (m.a \wedge m.i) \rangle$
, T8 applied to last deduction and MRP8

$NR = k \wedge \langle \exists m :: m.a \wedge m.i \rangle \mapsto NR = k + 1 \wedge \langle \exists m :: \neg(m.a \wedge m.i) \rangle$
, MRP7 $\wedge$ T9 $\wedge$ T5

$NR = k \wedge \langle \forall m :: \neg(m.a \wedge m.i) \rangle \mapsto NR = k + 1 \wedge \langle \exists m :: \neg(m.a \wedge m.i) \rangle$
, apply T6 to last two deductions

$NR = k \mapsto NR = k + 1$
, apply T7 to last two deductions and T10

$\langle \forall j : j \in \{0, 1, \ldots\} :: M = j \mapsto m < j \vee NR \geq \mathit{MaxRepairs} \rangle$
, apply definition of M to last deduction

$\mathit{true} \mapsto NR \geq \mathit{MaxRepairs}$
, apply T11 to last deduction □