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

LINKING SIMULATION MODEL SPECIFICATION AND PARALLEL EXECUTION THROUGH UNITY

Marc Abrams
Ernest H. Page
Richard E. Nance

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

ABSTRACT

A simulation specification language is used to describe a simulation model to a diverse audience, including programmers, managers, and clients. The advent of parallel programming imposes new demands on simulation specification languages. This paper presents early results of an attempt to bridge the gap between the requirements and philosophy of parallel programming (attention focused on crucial efficiency-related implementation details) and the requirements and philosophy of a simulation model specification (attention focused on correctly, and simply, describing model behavior). Chandy and Misra's UNITY is found to offer (or be amenable to) many of the features required to bride this gap.

1 INTRODUCTION

Many protocols for execution of simulation programs on various parallel computer architectures have been developed over the last ten years. Fujimoto (1990) discusses conservative and optimistic parallel protocols based on partitioning the simulation model.

The requirements for parallel execution place new demands on a "general purpose" simulation specification language. First, a specification language must be powerful enough to express the mapping of a simulation model to any protocol on any target parallel architecture. Second, the specification language should allow the mapping of a simulation model to a protocol and parallel architecture to be postponed until later stages of the development life cycle. Third, a simulation specification language should allow a single simulation model specification to be mapped to several, disparate protocols and algorithms. These requirements are particularly important in automated support of simulation model development.

This paper discusses a computational model and proof system called UNITY (Chandy, Misra 1988) useful for specification, and illustrates its use. For the illustration, the paper solves a well-known problem, simulation of tandem G/G/1 queues, using two disparate parallel simulation algorithms in the literature: a conservative algorithm, suited for asynchronous parallel architectures (Section 5); and the Greenberg, Lubachevsky, Mitrani (1990) method, suited for synchronous parallel architectures (Section 4). The exercise shows that UNITY allows specification of two different solution strategies, leading to two different parallel algorithms, to the same problem; therefore UNITY has promise as a parallel simulation specification language.

A companion paper (Abrams, Page, and Nance 1991) examines UNITY purely from the standpoint of its ability to express simulation models without regard to parallel execution, and presents a simulation model development methodology based on UNITY.

2 INTRODUCTION TO 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.

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 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.

The main contribution of UNITY is a computational model appropriate for a wide variety of computer architectures, and a proof system that allows proof of both *safety* and *progress* properties. A safety property of a program holds in all computation states, such as an *invariant* (defined later). 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 array A are 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.

2.1 Computational Model

"A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple assignment statements" (Chandy, Misra 1988, 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. (Chandy, Misra 1988, 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 simultaneous 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 computational model appears conventional if viewed as a set of state transition machines, where execution of an assignment statement corresponds to a transition.

UNITY's view of control flow is also unconventional. *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 writing parallel programs is hard because 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.

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 at early specification stages, and by including control flow in the form of a detailed execution schedule of assignments statements such that execution is efficient for a target architecture.

2.2 Programming Logic

UNITY contains a formal specification technique; its notation and logical relations are described next.

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–list* : *boolean–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 (1988, Ch. 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 *unless* $q \equiv \langle \forall s : s \; in \; F :: \{p \wedge \neg q\}\, s\, \{p \vee q\}\rangle$.

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

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. 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 program 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:

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

```
program sort
assign
   ⟨□ i : 0 ≤ i < N :: A[i], A[i+1] :=
      A[i+1], A[i] if A[i] > A[i+1]⟩
end {sort}
```

Figure 1: Sort Array A into Ascending Order.

A *<var-decl-list>* is a list of variable declarations in Pascal. The *<initial-list>* and *<asg-list>* are identical in syntax, except that "=" and ":=" are used, respectively. An *<asg-list>* has the form *<stmt>* □ *<stmt>* □ ⋯ □ *stmt*. The symbol "□" separates statements. A *<stmt>* has two forms: simple and quantified. Examples of simple statements are:

$x, y := y, x$	Multiple assignment: swap y and x.
$x := y \parallel y := 1$	Same as $x, y := y, x$.
$x := y$ if $y \geq 0 \sim -y$ if $y \leq 0$	Set x to absolute value of y.
$y := -y$ if $y \leq 0$	Set y to absolute value of y (identity assignment if $y > 0$).

A quantified statement has the form $\langle \parallel$ *var-list* : *boolean-expr* :: *assertion* $\rangle$. As an 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 (Chandy and Misra 1988, p. 32): Sort integer array $A[0..N]$, $N \geq 0$, in ascending order. 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 $i = 0, 1, \ldots, N$. This progress property is formalized in UNITY in the following assertion: $true \mapsto \langle \wedge i : 0 \leq i < N :: A[i] \leq A[i+1]\rangle$. Figure 1 contains a UNITY program meeting this specification.

2.4 Program Development by Composition

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

The union of two programs results from appending the code of both programs together. Syntactically, if P and Q denote programs, $P \Box Q$ denotes a program whose <*asg-list*> is the concatenation of the <*asg-list*> of both P and Q. Union is used in Section 5.5.

The superposition rule is used in the companion paper (Abrams, Page, and Nance 1991). 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." (Chandy, Misra 1988)

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:

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.

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. One approach is to layer the simulation model on the time flow mechanism, which in turn is layered on the parallel simulation protocol. Superposition is used to specify this layering.

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, 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" (Chandy, Misra 1988).

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

3 EXAMPLE: G/G/1 QUEUE

The remainder of the paper uses UNITY to specify a simulation of G/G/1 queues, both individually and in tandem. Before 1990, all parallel simulation protocols in the literature based on simulation model partitioning (Fujimoto (1990)) made each server a schedulable thread. Therefore a single G/G/1 queue could not be executed in parallel, and N queues in tandem could make use of a parallel architecture with at most N processors. In 1990, Greenberg, Lubachevsky, and Mitrani (GLM) (1990) eliminated these constraints, and showed how an unbounded number of processors could be used even with the single G/G/1 case. The GLM solution starts with a recurrence relation expressing the arrival and departure schedules in terms of the random variates for interarrival and service time.

A general purpose simulation specification language should permit one problem definition and then allow either the conventional or GLM solution strategies to be used. This paper shows that this is possible with UNITY. The GLM solution is given in Section 4, and the conventional solution is given in Section 5.

To formally state the problem, several definitions are required. An arrival sequence array of N jobs is an N element array where $A[i]$ is the simulation time at which the i^{th} job arrives, for $i = 1, \ldots, N$. A service time array is an N element array where $\delta[i]$ denotes the simulation time required to service the i^{th} job that arrives.

Given arrays A and δ, design a program that computes array D, where $D[i]$ (for $1 \leq i \leq N$) is the departure time of the i^{th} job that arrives, given that a first-come, first-served queueing discipline is used.

Formally, we wish to devise a program that computes N element array d and mean sojourn time S, such that:

$$\begin{aligned} &\textbf{invariant } FP \Rightarrow \\ &\quad d = D \ \wedge \\ &\quad S = \tfrac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: D[i] - A[i] \rangle \end{aligned} \tag{1}$$

$$\mathit{true} \mapsto FP. \tag{2}$$

Recall that FP denotes the fixed point of the program.

4 GLM SOLUTION

This section presents the solution devised by Greenberg, Lubachevsky, and Mitrani (1990), hereafter referred to as the GLM solution. The UNITY development given below is based on the all points shortest path algorithm given in Chapter 5 of Chandy and Misra (1988).

4.1 Solution Strategy

Let $D[0] = 0$. Greenberg, Lubachevsky, and Mitrani observe that array D is a solution to the following recurrence relation:

$$\langle \forall i :: D[i] \ = \ \max(\ D[i-1],\ A[i]\ ) + \delta[i] \rangle \tag{3}$$

Our initial solution approach is to initialize all elements of array d to zero. In our strategy, during simulation each element $d[i]$ either retains its value or increases in value until it reaches $D[i]$. Let $d[0] = 0$. Therefore the simulation simply executes the assignment $d[i] := \max(d[i-1], A[i]) + \delta[i]$ forever. UNITY's fairness rule along with invariant (1) implies that eventually $d[i]$ reaches $D[i]$ for all i.

The solution strategy does not impose any constraint on *how* the value of i is chosen, nor does it specify *what* processor should execute the assignment. Answering these questions tailors our solution strategy to a different computer architectures in the remainder of Section 4.

The informal strategy is now formalized. Because the value of $d[i]$ increases monotonically during simulation until it reaches the desired departure time $D[i]$, the following invariant holds:

$$\langle i : 1 \leq i \leq N :: d[i] \leq D[i] \rangle \tag{4}$$

The fixed point of this strategy holds when all the $d[i]$ remain unchanged, i.e.,

$$\begin{aligned} FP \equiv \langle i :: \\ &\quad d[i] = \max(d[i-1], A[i]) + \delta[i] \ \wedge \\ &\quad S = \tfrac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: d[i] - A[i] \rangle \ \rangle. \end{aligned} \tag{5}$$

Next we need to add to the specification an assertion that guarantees a fixed point is always reached. We show that if the simulation has not reached a fixed point, then at least one of the $d[i]$ increases. A common strategy in UNITY to prove such an assertion is to define a *metric*, a function of the state variables comprising the simulation model, and show that the metric must decrease in value if a fixed point has not been reached, and further that the metric has a minimum value. The metric we employ is the difference between the sum of all elements of array D and the sum of all elements of array d: $\mathit{sum} = \langle +i :: D[i] \rangle - \langle +i :: d[i] \rangle$. The metric is bounded below by zero because the value of each $d[i]$ cannot exceed $D[i]$ according to the invariant (assertion (4)). The progress condition is that the metric decreases if the state is not a fixed point. Formally, for all integers m,

$$\neg FP \wedge \mathit{sum} = m \mapsto \mathit{sum} < m. \tag{6}$$

4.2 Outline of Correctness Proof

Equations (4) to (6) represent the highest level specification of a program implementing a G/G/1 simulation using our solution strategy. We must formally verify that this specification correctly solves the problem. This requires showing that the invariant (assertion (1)) and progress condition (assertion (2)) of Section 3 are met by any program that satisfies conditions (4), (5), and (6).

4.3 A Simple Program

Program *GLM1* in Figure 2 embodies our solution strategy. Normally one must prove that program *GLM1* meets the specification of Section 4.1; the proof follows the outline given above. Program *GLM1* contains $N + 1$ assignment statements. It does not specify the *order* in which each of these assignments is executed. Due to the invariant (assertion (4)) and UNITY's unending selection of a statement to execute according to the fairness rule, all possible orders of assignment execution guarantee a correct solution, and the program eventually reachs FP.

```
program GLM1
  initially $\langle \| \, i :: d[i] = 0 \rangle$
  assign
    $\langle \Box i :: d[i] := \max(d[i-1], A[i]) + \delta[i] \rangle$
    $\Box S := \frac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: d[i] - A[i] \rangle$
end { GLM1 }
```

Figure 2: Basic GLM Program

4.4 Sequential Architecture

A refinement of *GLM1* suitable for execution on a sequential computer architecture is to explicitly increment i by one from an initial value of one to N; the result is program *GLM2*. To obtain program *GLM2*, add "$\| \, x = 1$" to the **initially** section and "$\| \, x := x + 1$ if $x < N$" to the **assign** section of program *GLM1*. Program *GLM2* requires $O(N)$ time to execute on a sequential processor.

4.5 Synchronous Parallel Architecture

Chandy and Misra (1988, p. 10) define a synchronous parallel architecture as "a fixed number of identical processors share a common memory that can be read and written by any processor. There is a common clock; at each clock tick, every processor carries out precisely one step of computation." The explicit incrementing of i from one to N in program *GLM2* yields an efficient algorithm for a sequential processor, but not for a synchronous parallel architecture.

A UNITY program can be mapped to a synchronous, parallel architecture by refining the program so that it contains a multiple assignment statement. Each processor can compute one of the right hand side expressions; all processors then synchronously make the assignment to the corresponding left hand side variable.

In program *GLM1*, we could change the "$\Box i$" to "$\| \, i$" without affecting program correctness to employ a multiple assignment statement that could be executed in constant time by $O(N)$ processors.

A more efficient solution is that of Greenberg, Lubachevsky, and Mitrani (1990). In the next refinement, the *max* function in program *GLM1* is rewritten. The result is program *GLM3*, shown in Figure 3. The program uses the GLM solution based on matrix multiplication where a binary maximum operator is the addition operation and addition is the multiplication operator.

The solution requires N matrices with 2×2 elements, denoted $M_1, M_2, \ldots, M_N$, and $N + 1$ vectors with 2 elements, denoted $V_0, V_1, \ldots, V_N$. If $V_i = \begin{bmatrix} y \\ z \end{bmatrix}$, then $V_i[0] = y$ and $V_i[1] = z$.

```
program GLM3
  initially
    $V_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
    $\Box \langle \| \, i :: M_i = \begin{bmatrix} \delta[i] & A[i] + \delta[i] \\ -\infty & 0 \end{bmatrix} \rangle$
  assign
    $\langle \Box i :: V_i = M_i \cdot M_{i-1} \cdots M_1 \cdot V_0 \rangle$
    $\Box \langle i :: d[i] = V_i[0] \rangle$
    $S := \frac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: d[i] - A[i] \rangle$
end { GLM3 }
```

Figure 3: Refinement Eliminating Max Function

Program *GLM3* requires $O(N^2)$ time on a single processor, or $O(N)$ time on $O(N)$ processors, which is no better than *GLM2* on a single processor. Therefore *GLM3* requires further refinement to reduce the running time on a synchronous parallel architecture below the running time of *GLM2* on a sequential architecture. The refinement, which is *GLM4*, uses the parallel prefix algorithm of Kruskal, Rudolph, and Snir (1985) to reduce the running time to $O(\log N)$ using $O(N)$ processors. This algorithm uses a combining tree to evaluate the to evaluate the assignment to V_i for all i in a single parallel operation. *GLM4* differs from *GLM3* by adding $j := 0$ to the **initially** section and changing the **assign** section to:

```
$\langle \; \langle \| \, i :: M_i = M_{i-2^j} \cdot M_i \rangle$
  $\| \, j := j + 1$ if $j < \lceil \log N \rceil - 1 \; \rangle$
$\Box \langle i :: d[i] = M_i V_i[0] \; \rangle$
$S := \frac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: d[i] - A[i] \rangle.$
```

To summarize this section, we presented a high level solution to the G/G/1 simulation in assertions (4) to (6). This strategy is refined to program *GLM2*, which requires $O(N)$ time on a sequential computer architecture, and to *GLM4*, which requires $O(\log N)$ time on $O(N)$ processors. UNITY is well suited to expression of the GLM solution strategy.

5 CONVENTIONAL SOLUTION

This section develops a solution suitable for execution by a conservative or optimistic parallel simulation protocol.

5.1 Specification Omitting Time

The solution strategy follows the four step program development methodology given in the companion paper (Abrams, Page, Nance 1991). This section describes a specification that captures the *order* of events that occurs in a single G/G/1 queue, ignoring the time at which events occur. Section 5.3 describes the addition of event timings to calculate departure sequence array d. Sections 5.4 and 5.5 present ways to map the program to a sequential and parallel architecture, respectively.

The specification consists of several objects: the queue, the server, and the jobs which pass through the system. These objects can be represented by their attributes. These attributes, or state variables, describe the state of the system by the values which they assume. The state of the server is either *busy* or *idle*. The queue has some nonnegative length. For convenience we employ variables $s.i$, $s.b$ and $q.n$, defined as:

$$\begin{aligned} s.i &\equiv (\mathit{server.state{=}idle}) \\ s.b &\equiv (\mathit{server.state{=}busy}) \\ q.n &\equiv (\mathit{queue.length{=}n}). \end{aligned}$$

The following sever assertions specification the G/G/1 simulation:

Q1: $s.i \wedge q = 0$ *unless* $s.i \wedge q > 0$
Q2: $s.i$ *unless* $s.b$
Q3: $s.i \wedge q > 0 \mapsto s.b$
Q4: $s.b$ *until* $s.i$
Q5: $q.n$ *until* $q.n + 1 \vee q.n - 1$
Q6: $q.n + 1 \wedge s.b$ *unless* $q.n + 1 \wedge s.i$
Q7: $q.n + 1 \wedge s.i \mapsto q.n$

The companion paper (Abrams, Page, Nance 1991) explains how to obtain these assertions mechanically from a state transition diagram. Figure 4 presents a program derived from the seven assertions.

5.2 Outline of Correctness Proof

To formally show that the program of Figure 4 meets the specification, proofs of each assertion in the specification must be carried out with respect to the program code. As an example, proof of assertion Q5 requires establishing $q.n$ *unless* $q.n+1 \vee q.n-1$; from

```
program CS1
declare
    q  : integer          {Number in queue }
    s  : (busy, idle)     {Status of the server}
    NS : integer          {Number served}
initially
    NS := 0 || q := 0 || s := idle
assign
    ⟨q := q + 1⟩
  □ ⟨s, q := busy , q − 1        if q > 0 ∧ s = idle ⟩
  □ ⟨s, NS := idle , NS + 1      if s = busy ⟩
end { MM1 }
```

Figure 4: Conventional Solution, Ignoring Timings

the definition of *until*, this requires showing that $\{q.n\}\, s\, \{(q.n) \vee (q.n+1 \vee q.n-1)\}$ holds for all statements s in the program. Finally, $q.n \mapsto q.n{+}1 \vee q.n{-}1$ must be established.

5.3 A Complete Solution

To meet the G/G/1 simulation problem specification (assertions (1) and (2)), program *Conv1* must be modified to compute the departure sequence array, d. The result is program *CS2*. Program *CS2* is obtained by following the methodology of the companion paper (Abrams, Page, Nance 1991). First we construct state transition diagram for each state variable (i, j, q, and $d[i]$). The diagrams are mapped to the set of assertions given in the Appendix using rules from the companion paper. Program *CS2* follows directly from the assertions.

5.4 Sequential Architecture

Mapping *CS2* to a sequential computer architecture simply requires specifying an execution schedule of the assignment statements in *CS2*. Any schedule meeting UNITY's fairness rule results in a correct implementation. Because both i and j assume all values from 1 to N, the program requires *O(N)* time to execute on a sequential processor, which means that the running time only differs by a constant factor from that of program *GLM2*.

5.5 Asynchronous Parallel Architecture

Conservative and optimistic simulation protocols can only use one processor to execute progam *CS2*. Therefore we consider P number of G/G/1 queues in tandem. Let *CS2*(y, z) denote program *CS2* with arrays y and z replacing arrays A and d, respectively.

```
program CS2
declare
    i, j     : integer  {Next arrival, departure index}
    q        : integer  {Number in queue, service}
    A[N]     : integer  {Arrival times}
    d[N+1]   : real     {Departure times}
    δ[N+1]   : real     {Residual service times}
    α[N+1]   : real     {Residual IATs}
initially
    $i := j := 1 \Box q := 0 \Box d[1] := 0 \Box \alpha[1] = A[1]$
    $\Box \delta[N+1] := \alpha[N+1] := \infty$
    $\Box \langle i : 1 < i \leq N :: \alpha[i] := A[i] - A[i-1] \rangle$
always $Q \uparrow = \alpha[i] < \delta[j] \Box Q \downarrow = \alpha[i] \geq \delta[j]$
assign
    $i, d[j], q := i+1, d[j]+\alpha[i], q+1$ if $q = 0$
  $\Box\ i, d[j], q, \delta[j] := i+1, d[j]+\alpha[i], q+1, \delta[j]-\alpha[i]$
      if $q > 0 \wedge Q \uparrow$
  $\Box\ j, d[j], d[j+1], q, \alpha[i] :=$
      $j+1, d[j]+\delta[j], d[j]+\delta[j], q-1, \alpha[i]-\delta[j]$
      if $q > 0 \wedge Q \downarrow$
  $\Box\ \langle S := \frac{1}{N} \cdot \langle +i : 1 \leq i \leq N :: d[i] - A[i] \rangle$
end { CS2 }
```

Figure 5: Conventional Solution, Including Timings

Then P tandem queues is denoted using the UNITY union operation from Section 2.4 as:

$$CS2(A, z_1) \Box CS2(z_1, z_2) \Box \cdots \Box CS2(z_{P-1}, z_P).$$

Mapping the composite program to an asynchronous parallel architecture requires a mapping of assignment statements to processors a refinement to eliminate the use of shared memory to store arrays z_i for $1 \leq i \leq P$. One mapping is to employ P processors, and map each instance of program *CS2* to a unique processor such that if array z_i is written by one instance of *CS2* and read by a second instance, then there is a memory which can be written to and read by the processor executing the writing and reading instances of *CS2*, respectively. The tandem simulation requires $O(N)$ time under this mapping.

6 EVALUATION OF UNITY

Model specification is the transformation of one system representation to another. Typically, a complex model specification requires a series of transformations, beginning with a conceptual view of a system and progressing through successive communicative forms; i.e., a communicative form is reproducible without error (see (Balci 1986) for a depiction of the model development life cycle). Although a specification in its own right, a program is the most concrete (free of abstraction) form, and is a representation of *how* model behavior is produced in addition to *what* is desired. This view thus characterizes the modeling (design) process as comprised of specification transformations that progressively resolve abstraction until the concrete model representation (program) is achieved.

Table 1, adapted from Barger and Nance (1986), present a checklist for dual assessment of (1) the properties of "good" specifications and (2) the capabilities of "good" specification languages. The identification of both language capabilities and specification properties suggests that an evaluation should utilize both the "tool" (language) and the product from using the tool.

UNITY is evaluated on a scale of zero to four, where zero means "Fails to provide this capability," two means "adequately provides this capability," and four means "provides this capability better than any known alternative."

UNITY has two major weaknesses when used as a simulation specification language. First, UNITY is based on a model of fair interleaving of statement execution, which provides no notion of time. Hence one cannot directly state and reason about properties of simulation time, which would be very useful in simulation modeling. (Sections 4 and 5 indirectly specify time through arrays α, δ, A, D, and d.) However the ability to reason about time is an area of active research, which may be embodied in future specification languages.

The second weakness arises because UNITY is a purely formal description tool, which limits the portion of the model development life cycle that it can be used with. UNITY is a narrow-spectrum language (Neighbors 1984), operating very near the implementation level of abstraction; i.e., the representation permits little abstraction beyond that realized with the programming language. In particular one would like an informal description to augment the formal description for managers and customers. In addition documentation of design decisions as a byproduct of the specification process is necessary. All of these could be added to UNITY, however.

These weaknesses aside, UNITY shows strong capability in the support of modular construction and the analysis of model completeness. The language is unambiguous and relatively easy to learn for individuals accustomed to formal notations; however, the proof procedure is not so readily mastered.

SPECIFICATION LANGUAGE CAPABILITIES	DESIRABLE SPECIFICATION PROPERTIES	SCORE	JUSTIFICATION
Model Organization:			
Encourages modularity	- Understandable - Information is localized - Easily modifiable	4.0	Building block approach strongly supported by superposition and union.
Encourages hierarchical description	- Suitable for many audiences - Presentable in varying levels of detail	2.0	Narrow spectrum language allowing hierarchy of low level descriptions that is inappropriate for some audiences.
Model Credibility:			
Use of application terminology allowed	- Understandable - Suitable for many audiences	2.0	Can use application terminology by supplying attributes. Application level references to time not permitted.
Documentation produced as a byproduct	- Understandable - Presentable in varying levels of detail	0.0	No documentation produced other than the low level UNITY assertions and program.
Assess specification completeness	- Analyzable	4.0	Real strength of UNITY is proof system.
Facilitates validation and verification	- Separates implementation and description details - Analyzable	3.0	Focuses close to implementation level. Inability to formally characterize "time" is a hindrance.
Specification Approach			
Permit both formal and informal constructs	- Understandable - Suitable for many audiences - Has environment description	1.0	Formal constructs only. Superposition, union allow enhancement and addition of objects without repeating proofs. Object redefinition may require new proofs.
Encourages use of developmental method	- Separates implementation and description details - Has environment description	2.0	Focus is at implementation level. Limited basis for system environment description (e.g., I/O behavior, interactive execution).
Accepts nonprocedural description	- Suitable for many audiences - Accommodates various system views	1.5	Accepts nonprocedural description at low level through assertions; readability is a problem.
Language Usability			
Easy to learn and use		0.5	Requires skill with formalisms.
Simple, precise, unambiguous syntax and semantics	Analyzable	3.5	Permits proof. Must develop intuition for logic relations and experience to use key theorems.
Full range of system behavior – static and dynamic		2.0	No way to formally reason about time. Otherwise adequate.
Independent of Simulation Programming Language	Separates implementation and description details	4.0	Algebraic, not operational specification; imposes no world view or programming language paradigm

Table 1: Assessment of UNITY as a Simulation Specification Language.

7 CONCLUSIONS

UNITY can handle both a state-transition based simulation specification (Section 5), which is the conventional parallel simulation program, as well as a data-flow based specification (Section 4), to its credit. For the G/G/1 problem, the data-flow view leads to a more efficient solution. This raises the question, do automation-based specification environments naturally bias their users to a state-transition view? Such a bias could complicate generation of efficient parallel simulation programs.

The state transition-based solution to the G/G/1 queue requires fourteen assertions, which looks excessive for a simple problem. The large number of assertions arises because our solution strategy explicitly specifies the *order* in which state transitions occur. (For example, our solution requires the state variables i, j, and q to always increase or decrease by one whenever their value changes.) Explicit specification of order leads to a large number of assertions. The UNITY philosophy of program development is to postpone decisions on order until later stages of program development. This implies that a state-transition based solution strategy is not well suited for eventual execution on a parallel architecture, or else UNITY is awkward to use for solutions with explicit sequencing. In contrast, the UNITY specification and program were quite natural for the data flow solution based on recurrence relations.

APPENDIX

Following are the UNITY assertions specifying a G/G/1 queue from which program *CS2* is derived.

Q1: $\langle i : 0 \leq i < N :: i = x \textit{ unless } i = x + 1 \rangle$
Q2: $\langle i : 0 \leq i < N :: i = x \wedge \neg(Q \uparrow \vee q = 0)$
$\textit{unless } i = x \wedge (Q \uparrow \vee q = 0) \rangle$
Q3: $\langle i : 0 \leq i < N :: i = x \wedge (Q \uparrow \vee q = 0)$
$\mapsto i = x + 1 \rangle$
Q4: $\langle j : 0 \leq j < N :: j = x \textit{ unless } j = x + 1 \rangle$
Q5: $\langle j : 0 \leq j < N :: j = x \wedge \neg(Q \downarrow \wedge q > 0)$
$\textit{unless } j = x \wedge (Q \downarrow \wedge q > 0) \rangle$
Q6: $\langle j : 0 \leq j < N :: j = x \wedge Q \downarrow \wedge q > 0 \mapsto$
$j = x + 1 \rangle$
Q7: $q = 0 \textit{ unless } q = 1$
Q8: $\langle q : 0 < q < N :: q = x \textit{ unless}$
$q = x - 1 \vee q = x + 1 \rangle$
Q9: $q = N \textit{ unless } q = N - 1$
Q10: $\langle q : 0 < q < N :: q = x \wedge Q \uparrow \mapsto q = x + 1 \rangle$
Q11: $\langle q : 0 < q < N :: q = x \wedge Q \downarrow \mapsto q = x - 1 \rangle$
Q12: $\langle j : 1 \leq j \leq N :: d[j] = k \textit{ unless}$
$d[j] = k + \delta[j] \vee d[j] = k + \alpha[i] \rangle$
Q13: $\langle j : 1 \leq j \leq N :: d[j] = k \wedge Q \downarrow \wedge q > 0$
$\mapsto d[j] = k + \delta[j] \rangle$
Q14: $\langle j : 1 \leq j \leq N :: d[j] = k \wedge (Q \uparrow \vee q = 0)$
$\mapsto d[j] = k + \alpha[i] \rangle$

REFERENCES

Abrams, M., E. H. Page, and R. E. Nance. 1991. Simulation Program Development by Stepwise Refinement in UNITY, *1991 Winter Simulation Conference.*

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

Barger, L.F. and R.E. Nance. 1986. Simulation Model Development: System Specification Techniques, Technical Report SRC-86-005, Systems Research Center, Virginia Tech, Blacksburg, VA.

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

R. M. Fujimoto. 1990. Parallel Discrete Event Simulation, *CACM*, 33 (10), pp. 30-53.

A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. 1990. Unboundedly Parallel Simulations Via Recurrence Relations, *SIGMETRICS 1990*, May, 1-12.

C. P. Kruskal, L. Rudolph, and M. Snir. 1985. The Power of Parallel Prefix. *IEEE Transactions on Computers*, C-34 (10), Oct. 1985.

Neighbors, J.M. 1984. The Draco Approach to Constructing Software from Reusable Components, *IEEE Transactions on Software Engineering* **SE-10**(5), September, 564-574.

AUTHOR BIOGRAPHIES

Refer to the paper, "Simulation Program Development by Stepwise Refinement in UNITY," elsewhere in these proceedings, for the author biographies.