Report Number: WUCS-92-34

1992-10-01

Architecture-Directed Refinement

Gruia-Catalin Roman and C. Donald Wilcox

As critical computer systems continue to grow in complexity, the task of demonstrating that they are correct, that is, guaranteed to operate without failure, becomes more difficult. For this reason, research in software engineering has turned to formal methods, i.e. rigorous approaches to demonstrating the correctness of software systems. Unfortunately, the formal methods currently used for concurrent systems do no provide a mechanism for expressing and manipulating non-functional constraints formally. In this paper, we show that one class of non-functional constraints, the target architectures, can be expressed using formal notation (the UNITY proof logic). We then use a mixture... **Read complete abstract on page 2.**

Architecture-Directed Refinement

Gruia-Catalin Roman and C. Donald Wilcox

Complete Abstract:

As critical computer systems continue to grow in complexity, the task of demonstrating that they are correct, that is, guaranteed to operate without failure, becomes more difficult. For this reason, research in software engineering has turned to formal methods, i.e. rigorous approaches to demonstrating the correctness of software systems. Unfortunately, the formal methods currently used for concurrent systems do no provide a mechanism for expressing and manipulating non-functional constraints formally. In this paper, we show that one class of non-functional constraints, the target architectures, can be expressed using formal notation (the UNITY proof logic). We then use a mixture of specifications and program refinements to derive a program which is demonstrably correct, both functionally and in its appropriateness for implementation on a specific machine.

School of Engineering & Applied Science

Architecture-Directed Refinement

Gruia-Catalin Roman
C. Donald Wilcox

WUCS-92-34

October 1992

To appear in IEEE Transactions on Software Engineering.

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

As critical computer systems continue to grow in complexity, the task of demonstrating that they are correct, that is, guaranteed to operate without failure, becomes more difficult. For this reason, research in software engineering has turned to *formal methods*, i.e., rigorous approaches to demonstrating the correctness of software systems. Unfortunately, the formal methods currently used for concurrent systems do not provide a mechanism for expressing and manipulating non-functional constraints formally. In this paper, we show that one class of non-functional constraints, the target architecture, can be expressed using a formal notation (the UNITY proof logic). We then use a mixture of specification and program refinements to derive a program which is demonstrably correct, both functionally and in its appropriateness for implementation on a specific machine.

Correspondence: All communications regarding this paper should be addressed to

Dr. Gruia-Catalin Roman
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu

1. Introduction

Increasingly, computers are used to control and monitor critical systems where failures are unacceptable. In many such systems, it is necessary to provide strong guarantees that each software component will function correctly. Because these systems are large, complex, and involve multiple computers, mere testing is not sufficient for demonstrating that the software is free of errors. Consequently, research in software engineering has been forced to consider more rigorous approaches to the specification, analysis, and construction of programs. Program derivation is a promising formal approach to constructing correct programs in which a program is created by mathematically manipulating formal specifications of the problem to be solved. An initial, abstract specification is gradually refined until it becomes sufficiently concrete as to suggest a direct realization in terms of some available programming language. The correctness of the final program is guaranteed by its construction.

Initial research on program derivation dealt exclusively with sequential programs and relied upon the weakest-precondition calculus [10]. Two general program construction strategies emerged from this work: *algorithm refinement* [1, 9, 14, 19, 20, 26] which is concerned with procedural abstractions and *data refinement* [15, 18, 21] which is concerned with data representations. As interest shifted into the realm of concurrent systems, it became necessary to consider new program derivation strategies.

Although this field is relatively new, several promising approaches to deriving concurrent systems have surfaced in recent years. Back and Sere, in their work on action systems [2], advocate a *program refinement* approach in which a series of correctness-preserving transformations are applied for the purpose of changing an initially large-grained (possibly sequential) program into a fine-grained, highly-concurrent one. Similar program-transformation ideas have been explored by several other researchers [11, 12, 13, 17], mainly in the context of the CSP model. In contrast, Chandy and Misra, in their work on UNITY [6], build on the legacy of algorithm and data refinement, working largely within the realm of specifications, deferring the writing of a program until the very end of the refinement process.

The UNITY proof logic combines the expressive power of the linear-time logics with the conceptual simplicity of Hoare-style predicates over a program's state. An initial specification consisting of assertions about both

the safety and the progress properties of the desired program is gradually refined until the construction of a correct program becomes straightforward. Both the data and the algorithm are refined along the way. To date there has been a great deal of program derivation work within the UNITY framework [7, 16]; Staskauskas, 1988 #712. Furthermore, the UNITY model has been extended by Roman and Cunningham [8, 23, 24] to accommodate increased levels of dynamism as well as additional programming paradigms, such as rule-based programming and dynamic synchrony. This extended model, *Swarm*, has been used to provide, for the first time, a formal framework for deriving parallel production systems [25]—it is worth noting that the resulting program derivation methodology combines elements of both specification and program refinement, an idea to which we will return later in this paper.

One significant shortcoming common to all of these approaches is that they exclude considerations regarding the target architecture from the formal framework while, at the same time, they make use of informal characterizations of the target architecture to justify individual refinements and to guide the derivation process toward and efficient solution. These *architectural constraints* have a significant impact on the direction taken by the refinement process; clearly, a proper solution to a given problem given a shared memory multiprocessor will be different than that for a message-passing network. However, because architectural constraints are stated informally, they cannot be factored (formally) into the derivation process and no one attempts to prove that the resulting program can actually be executed on the desired architecture.

While many designers may view the notion of formalizing architectural constraints with some skepticism—established formal derivation methods are yet to prove themselves in an industrial setting—the idea has important intellectual and practical implications. First, eventual tool support for design activities becomes possible only if they acquire a formal underpinning. Second, applications involving high degrees of reliability often require high performance levels achievable only through the utilization of specialized devices and novel architectures. The combination of reliability and performance is likely to render inadequate any ad-hoc reasoning about architectural constraints and to limit the applicability of current formal derivation methods.

The main contribution of this paper is to show that architectural constraints can be expressed formally using the same notational and logical system employed to specify behavior. Further, we show that the architectural constraints can be used to guide the process of constructing a program appropriate for the desired architecture. Our

approach starts with a formal specification of the program behavior, written using safety and liveness assertions over an abstract (global) state of the program. This specification is refined to produce an initial program which, while correct, does not necessarily map well onto the desired architecture. At this point formal specifications of the constraints imposed by the target architecture are added to the specification and used to guide a program refinement process leading to an architecture-specific program.

In the remainder of this paper, we elaborate on this methodology for deriving concurrent systems, with an example taken from distributed simulation. Section 2 outlines the design methodology in greater detail. Section 3 describes the Swarm notational system employed in this paper. In section 4, the example problem is described, and an initial, high-level specification is given. Section 5 traces the refinement of the specification to an initial abstract program. Section 6 illustrates the process of adding architectural constraints to the specification. For this purpose, two example architectures are selected: (1) a bus-based, message-passing architecture with specialized hardware for sharing information, and (2) a simple, unidirectional ring. Finally, sections 7 and 8 draw some conclusions.

2. Design Methodology

Our approach to the specification and design of concurrent systems is unique in its integration of architectural issues into the derivation process. The methodology is two-phased. *Specification refinement* is employed to construct a first program which is correct but architecture-independent. The specification method, refinement strategy, and notation are essentially those of UNITY. *Program refinement* is used to transform this program into another which satisfies the behavior specification as well as formally stated architectural constraints ignored during the specification refinement phase. The compatibility between the program and the target architecture is guaranteed by construction. Despite superficial similarities with techniques involving changes in the atomicity of program actions, the program refinement process presented here is new. The program transformations are not part of a standard, restricted repertoire. They are creative steps motivated by violations of the architectural constraints and may trigger limited re-verification of certain behavioral assertions. The amount of re-verification is kept small by carefully monitoring the relation between program actions and the assertions whose validity they may impact. A technique for formal specification and verification of architectural constraints is an integral part of this approach.

Initial specification. Like in UNITY, a program specification consists of safety and progress properties of the desired program. These safety assertions constrain the range of possible state transitions in which the program may engage; the progress assertions define state transitions that are required to take place. The specification is concerned only with the program behavior and makes no references to any non-functional constraints the program may have to meet (e.g., response time, reliability, cost, etc.). Furthermore, all assertions are stated in terms of a highly abstract state representation of the program. This is accomplished by substituting references to concrete data representations by predicates whose truth values are properly constrained and whose interpretation is given informally. For example, we can express the notion that some action α is executed at time t at node P by the predicate

$$\text{action}(P,\tau,\alpha).$$

Safety properties are specified using the **unless** relation as in

$$p \textbf{ unless } q$$

which asserts that if the program reaches a state in which the predicate p holds, p will continue to hold at least as long as q does not, which may be forever. Given the **unless** relation, one can easily introduce the notion of stability

$$\textbf{stable } p \equiv (p \textbf{ unless } \mathit{false})$$

which states that once p holds, it will continue to hold forever; and the concept of invariant

$$\textbf{inv. } p \equiv (\text{INIT} \Rightarrow p \wedge \textbf{stable } p)$$

which asserts that p holds initially and throughout the execution of the program. *INIT* characterizes the initial state of the program. Most often, the program initialization is not given explicitly but it is implied by the invariant properties.

Progress properties are specified using the **ensures** and **leads-to** (written "$\mapsto$") relations. The assertion

$$p \textbf{ ensures } q$$

requires that if the program reaches a state in which p holds, there is one specific action of the program which will be executed and will establish q while all other program actions either preserve the validity of p or are free to establish q. The assertion

$$p \mapsto q$$

simply states that if the program reaches a state in which p holds, it will eventually reach a state in which q holds. Unlike in the **ensures** relation, p is not required to hold until q is established, nor must there be one specific statement which establishes q. The **until** relation, defined as

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

is used to describe progress when p is required to hold until q is established, but no one specific statement is guaranteed to establish q atomically.

Architecture-independent specification refinement. Typical specifications tend to include many safety properties and relatively few progress properties. The former place constraints on the solution space and, for reasons of completeness, are rather detailed and numerous. The latter require that particular goal states must be reached but leave the details of how this is to be accomplished undefined. The purpose of specification refinement is to add sufficient detail on how progress is to be accomplished so as to make the writing of an appropriate correct program a trivial exercise. This means that relatively broad progress properties must be replaced by increasingly more specific ones. Changes in state representation often accompany these refinements and lead to more detailed formulations of the safety properties. Coupling invariants serve to relate predicates given in terms of one state representation to those given in terms of another, more concrete, representation. When the state representation and the progress properties match directly to data structures and statements in the target language, respectively, the transition from a specification to a program takes place.

Each specification refinement is a creative step motivated by design insights and carried out in a highly disciplined fashion. Although the syntactic form of a particular assertion may suggest a certain type of refinement,

such heuristics—which could ultimately lead to some form of automation—play only a secondary role in the refinement process today. In principle, the specification refinement could be biased towards a very specific architecture at the expense of rendering all other architectures inappropriate. In our methodology, however, the emphasis is on a general, architecture-independent solution. While this approach could be taken even in the absence of formal architectural constraints, it is rather the case that methodologies which treat architectural constraints informally need the target architecture to become the principal motivating factor behind the specification refinement, making an architecture-independent solution infeasible. As a result, each new architecture one might consider requires a new specification refinement. By attempting to generate first an architecture-independent program, our methodology makes it possible to perform the specification refinement once and then to use the same initial program as the basis for deriving multiple architecture-specific programs.

Architectural specification. It is a generally accepted fact that knowledge about the underlying architecture is required in order to construct an efficient or reliable program. It is also true that for applications that involve specialized hardware or network topologies, understanding the architecture is needed simply to write a program that can be executed, regardless of its performance. We define an architectural constraint to be any property a program must satisfy in order to execute on a given architecture. The absence of global data is one such constraint characterizing a distributed network. Each constraint defines a class of acceptable programs. An architectural specification consists of a number of constraints which must be satisfied simultaneously.

Formal specification of architectural constraints—not to be confused with hardware specification languages—is made difficult by the fact that one must define an entire class of programs having unknown behaviors. The task is further complicated by our desire to specify architectural constraints in the same assertional style and notation being employed in dealing with behavior specifications—simply called program specifications. The use of a single common notation for both program specifications and architectural constraints is motivated by our ultimate goal of integrating architectural considerations in the program derivation process.

The specification technique we describe in this paper relies on the introduction of auxiliary variables whose purpose is to capture certain salient relationships between a program and the architecture on which it is executed.

Architectural constraints are specified as assertions involving the auxiliary variables. The absence of globally shared data could be stated as

inv. access(i,k) $\Rightarrow$ i=k

where the predicate *access(i,k)* is to be interpreted to mean a statement on processor *i* accessed data on processor *k*. Such a fact may be stated without having the program at hand. Once the program is known, however, its statements may be augmented with auxiliary variables in accordance with a pre-defined set of rules and the invariant above can be in fact proven to hold for the particular program. The augmentation rules can be defined formally but those we use in this paper are simple enough to be described informally without undermining the rigorous nature of the presentation.

Architecture-driven program refinement. Because of its generality, the program generated at the conclusion of the specification refinement is unlikely to satisfy the architectural constraints that may be involved in the problem at hand. Detected violations of the architectural constraints are then used to guide the program refinement. In the simplest terms, our approach requires one to keep a checklist of constraints which are not satisfied by the program in its current form, to pick one from the list and fix it, updating the list accordingly. The process is complete when all the constraints are met. Because the entire process is performed within the formal framework, with proofs when necessary, the final program is known to be both correct and implementable on the target architecture.

Program refinements may involve changes in data representation, in the granularity of program statements, and in the allocation of statements and data to architectural components. Each refinement may trigger re-verification against both the program specification and the architectural constraints. In practice, modular design and careful staging of the program refinements can limit to a significant degree the extent of such re-verifications.

3. Notation

In this paper, we will be using *Swarm* [23], an extension to UNITY to reason about and write programs. Swarm belongs to a class of languages and models that use tuple-based communication. Other languages and models in this class are Linda [4], Associons [22], and GAMMA [3]. Two features of Swarm that are of particular impor-

tance for this paper are (1) its UNITY-like proof logic, and (2) the ease with which it can accommodate a variety of programming paradigms (e.g., shared variables, message passing, rule-based) and architectures (e.g., synchronous, asynchronous, reconfigurable, etc.). This section briefly describes the Swarm notation and proof logic. For a more complete discussion, the reader is referred to [8, 23, 24].

3.1. The Swarm Programming Notation

The primary means for communication among the concurrent components of a Swarm program is a common, content-addressable data structure called a *shared dataspace*. Elements of the dataspace may be examined, inserted, or deleted by programs. The model partitions the dataspace into three subsets: **TPS**, the *tuple space* (a finite set of data tuples), **TRS**, the *transaction space* (a finite set of transactions), and **SC**, the *synchrony relation* (a symmetric relation on the set of all possible transactions). A Swarm *transaction* denotes an atomic transformation of the dataspace. Instances of transactions may be created dynamically by an executing program.

A Swarm program begins execution from a specified initial dataspace. On each execution step, a transaction is chosen nondeterministically from the transaction space and executed atomically. This selection is *fair* in the sense that every transaction in the transaction space will eventually be chosen. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its execution. Program execution continues until there are no transactions remaining in the dataspace.

The synchrony relation is a relation over the set of possible transaction instances. This relation may be examined and modified by programs in the same manner as the tuple and transaction spaces. The synchrony relation affects program execution as follows: whenever a transaction is chosen for execution, all transactions in the transaction space which are related to the chosen transaction by (the transitive closure of) the synchrony relation are also chosen; all of the transactions that make up this set, called a *synchronic group*, are executed as if they comprised a single transaction.

Next we illustrate the Swarm programming notation using a toy example, a program in which at most N timers are incremented in lockstep fashion. Each timer *i* is incremented modulo some overflow value *ovr(i)*. A timer may be brought on line any time during the computation, but eventually all timers mark time in step with each other. To accomplish this, all timers are reset to zero whenever any one of them is reset to zero. As a result, all active timers count modulo $m=(min\ i : 1 \le i \le N \wedge active(i) :: ovr(i))$.[1]

Construct a timer. We begin by first considering the case of a simple timer with identifier *i*. We can represent the current state of this timer as a tuple *time(i, v)*, where *v* is the timer's current value. The transaction which increments and resets the timer is *Timer(i,ovr(i))*. A timer is activated by inserting in the dataspace (initially or during the computation) the corresponding data tuple and transaction.

Define the timer's behavior. A transaction stored in the dataspace is simply a name for an atomic transformation of the dataspace. The transaction's behavior is defined separately as the composition of one or more *subtransactions*. A subtransaction consists of a dataspace *query*, which binds some set of existentially quantified local variables whose scope extends over the entire subtransaction, followed by an *action* which modifies the contents of the dataspace by inserting or removing entries if the query succeeds. (Notationally, the query and action are separated by "→", we use the comma for logical *and* (∧), and sub-transactions are separated by "‖".) By definition, deletions are performed before insertions. The query can be any arbitrary predicate over the dataspace, similar to a Prolog goal, and may check for the presence (or absence) of specific entities in the dataspace. The semantics of transaction execution are similar to those for a single subtransaction, except that the queries for all subtransactions are evaluated in parallel, followed by the deletions and then the insertions appearing in the actions of those sub-transactions whose queries succeeded.

In our example, we can specify the behavior of an individual timer by introducing the following transaction type definition (the reason for dividing the first two transactions will be made clear later):

Timer(id, MAX) ≡
t : time(id, t), t ≥ MAX → **skip**
‖ t : **OR**, time(id, t) → time(id, t)†, time(id, 0)
‖ t : **NOR**, time(id, t) → time(id, t)†, time(id, t+1)
‖ **TRUE** → Timer(id, MAX)

The first subtransaction consists of a regular query (a query which does not use any special predicates[2]) which checks whether or not the timer needs to be reset and has a null action (*skip*). The variable t, which is local to this subtransaction, is bound by finding in the dataspace a tuple of type *time* whose first component contains the constant *id*. The success of the first subtransaction is communicated to the second subtransaction via the special predicate **OR**, which succeeds whenever any regular query executed in parallel evaluates to *true*. As a result, the second subtransaction resets the timer by deleting the tuple *time(id,t)*, independently found by its own query, and by inserting the tuple *time(id,0)*. Similarly, the third subtransaction uses **NOR** (i.e., not **OR**) to determine if the timer can be incremented by one unit. Finally, the fourth subtransaction recreates the timer (which otherwise would be implicitly deleted). The special predicate **TRUE** (which always succeeds) ensures that the query associated with this subtransaction becomes a special query and is therefore not considered when **OR** and **NOR** are evaluated.

Establish lockstep execution. The requirement for lockstep execution can be expressed in Swarm using the third type of dataspace entity, the *synchrony relation*. Two timers i and j can be made part of the same synchronic group by inserting into the dataspace the following synchrony relation entry:[3]

Timer(i,ovr(i))~Timer(j,ovr(j))

Recall that a set of transactions present in the dataspace and closed under the reflexive transitive closure of the synchrony relation is called a *synchronic group*, and that whenever a transaction is selected for execution, the entire synchronic group to which it belongs is executed, and all the subtransactions for all transactions in the group are executed together as if they were part of a single larger transaction. An interesting consequence of these semantics is that the special predicates are now evaluated with respect to the regular queries of the entire synchronic group—we had this in mind when we decided to use special queries in the definition of *Timer* above. Consequently, the special predicate **OR** evaluates to true whenever the query of the first subtransaction in either *Timer(i,ovr(i))* or *Timer(j,ovr(j))* succeeds, indicating that both timers must be reset.

3.2. The Swarm Proof Logic

Properties in the Swarm logic are expressed in terms of predicates over the global system state. Safety properties are given in terms of **unless**, and progress in terms of **ensures** (for atomic transitions), and $\mapsto$ (read

leads-to) for transitions of a larger grain. Other concepts, such as invariance, can be expressed in terms of these basic properties. Figure 1 summarizes the properties which we will use in this paper. The main difference between the UNITY and Swarm logics is the addition of dynamic control constructs; whereas in UNITY the set of statements in a program is fixed, the transactions in a Swarm program are drawn from a (possibly) infinite set, and the collection of transactions currently instantiated can changes during program execution. Similarly, the synchrony relation provides for dynamically changing the size of the atomic actions within a program by changing the form of the relation. These differences show up only in the formal execution model and in the definitions of the basic relations (**unless**, **ensures**, and **leads-to**), the theorems proven within the UNITY model continue to hold for Swarm. It should be clear to the reader that the Swarm inference rules reduce to those of UNITY when the synchrony relation is empty and the set of instantiated transactions remains constant.

1. $\{ p \}\ s\ \{ q \}$
   Given predicates p and q and a synchronic group s, this assertion (also called a "Hoare triple") holds if in every state satisfying the precondition p, the execution of s results in a state satisfying the postcondition q.

2. p **unless** q
   If p is *true* at some point in the computation and q is not, then executing any synchronic group either maintains p or establishes q, i.e.,
   $$\{ p \wedge \neg q \}\ s\ \{ p \vee q \}$$
   unless constrains the valid set of state transitions within a program.

3. **inv.** $p \equiv (INIT \Rightarrow p) \wedge (p$ **unless** *false*$)$
   The property p is *true* at all points in the computation, i.e., invariant.

4. p **ensures** $q \equiv p$ **unless** $q \wedge \langle \exists\, s : s \in \mathbf{SG} :: \{ p \wedge \neg q \}\ s\ \{ p \vee q \} \rangle$
   If $p \wedge \neg q$ is *true*, there exists a transaction t such that every synchronic group s containing t will establish q when executed. The fairness assumption guarantees that s will eventually be selected.

5. $p \mapsto q$
   This, read p **leads to** q, means that once p becomes *true*, q will eventually become *true*, but p is not guaranteed to remain *true* until q becomes *true*. Note that $\mapsto$ is transitive, whereas **ensures** is not.

6. p **until** $q \equiv (p \mapsto q) \wedge (p$ **unless** $q)$
   p **until** q is a special case of the **leads-to** relation, which requires that p continue to hold as long as q does not hold. Unlike **ensures**, there is no requirement that the transition from p to q take place in one atomic step.

7. p **detects** $q \equiv (p \Rightarrow q) \wedge (q \mapsto p)$
   Property p can be used to determine that some, typically more complex, property q has been satisfied.

Figure 1: Notation used in the Swarm proof logic.

4. Initial Specification

To illustrate our methodology, we will be using an example that was inspired by previous work on distributed simulation [5]. Consider a network of sequential nodes which can exchange messages over communication links. Each node executes a sequence of actions. Each action consists of the retrieval of pending messages from other nodes, the updating of some local data and the sending of messages to other nodes over the links. We assume that the links are error-free. We wish to specify and design a distributed program which will simulate this system. To avoid any confusion, we will limit the use of the terms "network" and "node" to refer to the simulated system, and reserve the term "program" for the simulator.

4.1. State Representation

For specification purposes, it is convenient to assume an absolute global time. The predicate *gclock(T)* is used to denote the fact that the current time is T. The network's state can be characterized in terms of the *actions* which are to be executed, the *messages* which are to be delivered, and the local *state* of each node. The following predicates are used to represent this information:

state(P,σ)	The current data state of node P is σ.
action(P,τ,α)	Action α will be executed at time τ at node P. We define the special action $\perp$ to be the *halt* action, i.e., when *action(P,τ,⊥)* is true, processor P is terminated. Messages for a terminated process may not remain undelivered.
message(P,Q,τ,μ)	A message from P to Q with content μ will be delivered at time τ.

The predicates introduced so far define an abstract state representation for the program. Since not all states are acceptable, several invariants are introduced which serve to constrain the state space in a reasonable way (all free variables are assumed to be universally quantified):

F1: *There is exactly one gclock value*
inv. $\langle \Sigma\, T : \text{gclock}(T) :: 1 \rangle = 1$

F2: *Each process has a unique state*
inv. $\langle \Sigma\, \sigma : \text{state}(P,\sigma) :: 1 \rangle = 1$

F3: *Each node executes one action at a time*
inv. $\langle \Sigma\, \tau,\alpha : \text{action}(P,\tau,\alpha) :: 1 \rangle = 1$

F4: *Actions are never "in the past"*
inv. $\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(T) \Rightarrow T \leq \tau$

F5: *Message values are unique across the network*
inv. $\text{message}(P,Q,\tau,\mu) \wedge \text{message}(P',Q',\tau',\mu') \Rightarrow ((P,Q,\tau) = (P',Q',\tau') \Leftrightarrow \mu = \mu')$

F6: *Messages are never "in the past"*
inv. $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \Rightarrow T \leq \tau$

An alternate, but equivalent, reformulation of F4 and F6 is useful:

F4': **inv.** $\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(T) \Rightarrow T \leq \langle \textbf{min}\ P',\tau',\alpha' : \text{action}(P',\tau',\alpha') \wedge \alpha' \neq \perp :: \tau' \rangle$

F6': **inv.** $\text{message}(P,Q,\tau,\alpha) \wedge \text{gclock}(T) \Rightarrow T \leq \langle \textbf{min}\ P',Q',\tau',\alpha' : \text{message}(P',Q',\tau',\alpha') :: \tau' \rangle$

We say that an action is *enabled* when the system clock has reached the action activation time, and there are no messages to be delivered at the current time. A processor is *halted* if its action is $\perp$. Formally,

$$\text{enabled}(P,\tau,\alpha) \equiv \text{action}(P,\tau,\alpha) \wedge \text{gclock}(\tau) \wedge \langle\, \forall\, Q,\mu :: \neg\text{message}(Q,P,\tau,\mu) \,\rangle \wedge \alpha \neq \perp$$

$$\text{halted}(P) \equiv \langle\, \exists\, \tau :: \text{action}(P,\tau,\perp) \,\rangle$$

Since many details of the network's computation are not relevant to the simulation program, we encapsulate them using several functions assumed to be available to the program:

$u(P,\sigma,\mu)$	returns the state of node P to when the absorption of a message μ is absorbed in state σ.
$e(P,\tau,\sigma,\alpha)$	returns the time when the action immediately following α will be executed given that α is executed on node P at time τ in state σ. The function e is strictly monotonic with respect to the argument τ.
$a(P,\sigma,\alpha)$	returns the name of the action which will be executed at node P subsequent to the completion of the action α in state σ.
$c(P,Q,\sigma,\alpha)$	is *true* if node P sends a message to node Q as a result of executing the action α in state σ.
$l(P,Q,\tau,\sigma,\alpha)$	returns the delivery time for a message sent by node P to node Q as a result of executing action α in state σ. Because a message is sent at the completion of the initiating action, $l(P,Q,\tau,\sigma,\alpha)$ must exceed $e(P,\tau,\sigma,\alpha)$.
$v(P,Q,\sigma,\alpha)$	returns the contents of the message sent to node Q by node P when executing the action α in state σ.
$s(P,\sigma,\alpha)$	returns the new state of node P resulting from executing the action α in state σ.

We now turn our attention to the problem of describing the assumptions we make about the behavior exhibited by the network to be simulated. As a means of organizing ourselves, we discuss valid changes to the different components of the network state separately. Properties involving several state components are discussed when first encountered.

4.2. Time

In the real world, time advances one unit at a time. If we impose this constraint in our specification, we will make it impossible for the program to advance time in larger increments whenever there is no network activity at any node. To avoid this unnecessarily strong condition, we will simply restrict time from moving backwards, leaving the increment unspecified:

F7: $\text{gclock}(\tau)$ **unless** $\langle\, \exists\, \tau' : \tau' \geq \tau+1 :: \text{gclock}(\tau') \,\rangle$

This formulation also allows the movement of time to cease (because it is a safety property and not a liveness property) when the simulation is "finished."

4.3. Messages

Upon being created as a result of executing an action, a message travels through the network in some unspecified manner until its delivery time (F9). Messages are delivered in arbitrary order (we consider a message *delivered* when it is deleted by the receiving processor), and the delivery of a message results in an atomic update of the state of the recipient (F10). Messages can be created only by executing an action (F8, F11).

F8: *Messages are only created by executing actions*
$\text{action}(P,\tau,\alpha) \wedge \langle$ **set** $Q,\tau',\mu : \text{message}(P,Q,\tau',\mu) :: \mu \,\rangle = M$
unless
$\neg\text{action}(P,\tau,\alpha) \vee \langle$ **set** $Q,\tau',\mu : \text{message}(P,Q,\tau',\mu) :: \mu \,\rangle \subset M$

F9: *Messages exist until their delivery time*
$\text{message}(P,Q,\tau,\mu)$ **unless** $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

F10: *Messages are absorbed at the time of their delivery*
$\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$ **unless** $\neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

F11: *Messages are not created at the current time*
$\neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$ **unless** $\neg\text{gclock}(\tau)$

F12: *Messages exist until incorporated into the state of the receiving node*
$\text{state}(P,\sigma) \wedge \text{gclock}(\tau) \wedge \langle$ **set** $Q,\mu : \text{message}(Q,P,\tau,\mu) :: \mu \,\rangle = M \wedge M \neq \{\}$
unless
$\langle\, \exists\, \mu : \langle$ **set** $Q,\mu' : \text{message}(Q,P,\tau,\mu') :: \mu' \,\rangle = M\text{-}\{\mu\} :: \text{state}(P,u(P,\sigma,\mu)) \,\rangle$

4.4. Actions

Although it is simpler to think of actions as being atomic and instantaneous, they in fact have a duration which should be modeled in the simulation. A simple way to capture this is to disallow the subsequent action from

executing until the first is completed. This allows us to view an action as waiting for its execution time (F13, F14). We require terminated nodes to remain so forever (F16) — and assume that messages whose delivery time coincides with the execution time of an action are absorbed before the action executes.

F13: *An action continues to exist until it is time to execute it*
$\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp$ **unless** $\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(\tau)$

F14: *An action continues to exist until it is enabled*
$\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(\tau)$
unless
$\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(\tau) \wedge \langle\, \forall\, Q,\mu :: \neg\text{message}(Q,P,\tau,\mu) \,\rangle$

F15: *An enabled action continues to exist until executed atomically*
$\text{state}(P,\sigma) \wedge \text{enabled}(P,\tau,\alpha)$
unless
$\text{action}(P,e(P,\sigma,\alpha),a(P,\sigma,\alpha)) \wedge$
$\text{state}(P,s(P,\sigma,\alpha)) \wedge$
$\langle\, \forall\, Q : c(P,Q,\sigma,\alpha) :: \text{message}(P,Q,l(P,Q,\tau,\sigma,\alpha),v(P,Q,\sigma,\alpha)) \,\rangle$

F16: *A terminated node remains terminated*
stable $\text{halt}(P)$

4.5. State

The only events which can change the state of a processor are the delivery of a message or the execution of an action. Note that these transitions have already been described (F12 and F15). All that remains is to forbid any other changes to the state.

F17: *The state does not change in the absence of work to do*
$\text{state}(P,\sigma)$ **unless** $\text{state}(P,\sigma) \wedge \langle\, \exists\, Q,\tau,\alpha,\mu : \text{gclock}(\tau) :: \text{enabled}(P,\tau,\alpha) \vee \text{message}(Q,P,\tau,\mu) \,\rangle$

4.6. Progress

The final section of the initial specification contains the liveness properties which describe the state transitions which are required by the system. Only two progress properties are needed to describe the required state transitions, messages must be delivered, and actions must be executed:

F18: *Messages must be delivered*
$\text{message}(P,Q,\tau,\mu) \mapsto \neg\text{message}(P,Q,\tau,\mu)$

F19: *Actions must be executed*
$\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \mapsto \neg\text{action}(P,\tau,\alpha)$

Other aspects of the network's behavior (such as moving time forward) are implied by combining these progress properties with the earlier stated safety properties. For example, if an action has an execution time in the future, then the clock must eventually move forward by properties F13 and F19.

This completes the specification of the network behavior. By carefully abstracting away irrelevant details, we are able to generate a specification which is both concise and clear.

5. Specification Refinement

In this section we refine the behavior specification up to the point where program generation becomes trivial. No architectural constraints are considered at this point. The refinement process is guided by the need to discover a series of simple state transitions that realize the progress conditions. More specifically, most of the refinements are involved with generating a precise specification of when and how the simulation clock should be incremented. The steps used here are not generic but specific to the problem. We view program derivation as a creative step, and not a mechanical substitute for design.

We use the notation

$p \vdash\!\!\!- q$

to mean "q can be proves from the specification which results from replace q with p."

5.1. Add specificity to the processing of messages and actions.

The first refinement is suggested by properties F9–F10 and F13–F15, which characterize the life-cycle of messages and actions into a series of state transitions. By combining this information with the progress properties F18 and F19, it is clear that (1) messages must be held until the time of delivery and no further; and (2) actions must be held until all messages having delivery time equal to the start of the action have been processed, and the effect of the action must be atomic.

Refinement. F18 is replaced by

F18.1: $\text{message}(P,Q,\tau,\mu) \mapsto \text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

F18.2: $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau) \mapsto \neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

which are analogous to F9 and F10; F19 is replaced by

F19.1: $action(P,\tau,\alpha) \wedge \alpha \neq \perp \mapsto action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau)$

F19.2: $action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau)$
$\mapsto$
$action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau) \wedge \langle\, \forall\, Q, \mu :: \neg message(Q,P,\tau,\mu) \,\rangle$

F19.3: $enabled(P,\tau,\alpha) \wedge state(P,\sigma)$
$\mapsto$
$action(P,e(P,\sigma,\alpha),a(P,\sigma,\alpha)) \wedge$
$state(P,s(P,\sigma,\alpha)) \wedge$
$\langle\, \forall\, Q : c(P,Q,\sigma,\alpha) :: message(P,Q,l(P,Q,\tau,\sigma,\alpha),v(P,Q,\sigma,\alpha)) \,\rangle$

which are analogous to F13-F15.

Proof Obligations. We are required to show:

$F18.1, F18.2 \vdash F18$

$F19.1 \wedge F19.2 \wedge F19.3 \vdash F19$

The proof follows from the transitivity of the **leads-to** relation.

Refined Specification.

Safety: F1-F17

Progress: F18.1, F18.2, F19.1, F19.2, F19.3

5.2. Decouple time movement from message and action processing.

The movement of time is required only by the presence of messages to be delivered and actions to be executed. Properties F18.1 and F19.1 indicate this, but they couple the movement of time with the continued presence of messages or actions. These two concerns may be decoupled very easily in light of the fact that F9 and F13 are still part of this specification.

Refinement. F18.1 and F19.1 are replaced by:

F18.1.1: $message(P,Q,\tau,\mu) \mapsto gclock(\tau)$

F19.1.1: $action(P,\tau,\alpha) \wedge \alpha \neq \perp \mapsto gclock(\tau)$

Proof Obligations.

F18.1.1 $\vdash$ F18.1

F19.1.1 $\vdash$ F19.1

The proof is immediate from the application of the progress-safety-progress (PSP) rule [6].

Refined Specification.

Safety: F1-F17

Progress: F18.1.1, F18.2, F19.1.1, F19.2, F19.3

5.3. Place lower bound on time increment interval

There are two constraints on the mechanism by which simulation time is moved forward, and neither is reflected in F18.1.1 or F19.1.1. First, time must move forward in steps of at least one unit (by F7), and it cannot move beyond the earliest time in which a message exists to be delivered (by F4), or an action to be executed (by F6). The refinement is in two steps: in the first, we fold the lower bound into the progress properties. The second refinement, which adds the upper bound, is deferred until later, when the proof of its correctness is easier.

Refinement. Properties F18.1.1 and F19.1.1 are replaced by:

F18.1.1.1: $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \wedge T < \tau \mapsto \langle\, \exists\, T' : T' \geq T+1 :: \text{gclock}(T') \,\rangle$

F19.1.1.1: $\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge \text{gclock}(T) \wedge T < \tau \mapsto \langle\, \exists\, T' : T' \geq T+1 :: \text{gclock}(T') \,\rangle$

Proof Obligations.

F18.1.1.1 $\vdash$ F18.1.1

Proof outline. We can prove the following lemma using PSP with F9 and F18.1.1.1.

L1: $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \wedge T < \tau$
$\mapsto$
$\text{message}(P,Q,\tau,\mu) \wedge (\langle\, \exists\, T' : T' \geq T+1 :: \text{gclock}(T') \,\rangle \vee \text{gclock}(\tau))$

Using invariant F6 with L1, we can prove L2:

L2: $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \wedge T < \tau$
$\mapsto$
$\text{message}(P,Q,\tau,\mu) \wedge (\langle\, \exists\, T' : \tau \geq T' \geq T+1 :: \text{gclock}(T')\, \rangle \vee \text{gclock}(\tau))$

This is precisely the left-hand side of the induction principle for **leads-to** [6], where the well-founded metric is the difference between the delivery time of a *message* and the current *gclock*. Thus, we have:

L3: $\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \wedge T < \tau \mapsto \text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

and

$\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(T) \wedge T = \tau \Rightarrow \text{gclock}(\tau)$

from which we conclude F18.1.1. The proof for F19.1.1 is similar. □

Refined Specification.

Safety: F1-F17

Progress: F18.1.1.1, F18.2, F19.1.1.1, F19.2, F19.3

5.4. Unify clock control

The similarity in the forms of properties F18.1.1.1 and F19.1.1.1 (and F4 and F6), suggests that messages and actions play identical roles with respect to clock movement. This leads us to consider a refinement in which each pair of properties is replaced by a single property which replaces them. The first refinement rewrites F18.1.1.1 and F19.1.1.1, the next handles F4 and F6.

Refinement 1. F18.1.1.1 and F19.1.1.1 are replaced by

F20: $\text{gclock}(T) \wedge (\text{message}(P,Q,\tau,\mu) \vee (\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp)) \wedge \tau > T$
$\mapsto$
$\langle\, \exists\, T' : T' \geq T+1 :: \text{gclock}(T')\, \rangle$

which is simply the disjunction of F18.1.1.1 and F19.1.1.1.

Proof Obligations.

F20 $\vdash$ F18.1.1.1

F20 ⊢ F19.1.1.1

Proof Outline. To show that F20 implies F18.1.1.1, we must prove that the left-hand side of F18.1.1.1 implies the left-hand side of F20, and that the right-hand side of F20 implies the right-hand side of F18.1.1.1, both of which are obvious by inspection. The second proof is the same.

Refinement 2. F4 and F6 (or F4' and F6') are replaced by

F21: **inv.** ((action(P,τ,α) ∧ α ≠ ⊥) ∨ message(P,Q,τ,μ)) ∧ gclock(T) ⇒ T ≤ τ

F21': **inv.** ((action(P,τ,α) ∧ α ≠ ⊥) ∨ message(P,Q,τ,μ)) ∧ gclock(T)
⇒
T ≤ ⟨ **min** P,Q',τ',α',μ' : (action(P,τ,α) ∧ α ≠ ⊥) ∨ message(P,Q,τ,μ) :: τ ⟩

Proof Obligations.

F21 ⊢ F4

F21 ⊢ F6

The proof for the refinement of F4 and F6 is immediate.

Refined Specification.

Safety: F1-F3, F5, F7-F17, F21

Progress: F18.2, F19.2, F19.3, F20

5.5. Refine mechanism for message delivery

Property F18.2 requires only that messages be delivered before time moves forward, but does not provide any hints into the delivery mechanism. Property F12 provides the necessary insight, in that it requires that messages be delivered and absorbed in a single atomic step. This suggests the obvious refinement of replacing F18.2 with a progress property having the same form as F12.

Refinement. F18.2 is replaced with

F18.2.1: gclock(τ) ∧ ⟨ **set** Q,μ' : message(Q,P,τ,μ') :: μ' ⟩ = M ∧ M ≠ {} ∧ state(P,σ)
↦
⟨ ∃ μ : ⟨ **set** Q,μ' : message(Q,P,τ,μ') :: μ' ⟩ = M-{μ} :: state(P,u(P,σ,μ)) ⟩

Proof Obligations.

F18.2.1 ⊢ F18.2.

Proof Outline. The proof makes use of the induction principle for **leads-to**, where the well-founded set is the set of messages which remain to be delivered at the current time. The induction principle requires that we prove the following lemma, which implies F18.2:

L4: $\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge$
$\langle\ \textbf{set}\ P',Q',\mu' : \text{message}(P',Q',\tau,\mu') \wedge \mu \neq \mu' :: \mu'\ \rangle = M$
$\mapsto$
$(\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ P',Q',\mu' : \text{message}(P',Q',\tau,\mu') \wedge \mu \neq \mu' :: \mu'\ \rangle \subset M) \vee$
$(\text{gclock}(\tau) \wedge \neg\text{message}(P,Q,\tau,\mu))$

By applying PSP to F18.2.1 and F10, we can conclude the following:

$\text{gclock}(\tau) \wedge \langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle = M \wedge M \neq \{\} \wedge \text{state}(P,\sigma) \wedge$
$\text{message}(P,Q,\tau,\mu)$
$\mapsto$
$(\langle \exists\ m :\langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle = M\text{-}\{m\} :: \text{state}(P,u(P,\sigma,m))\ \rangle \wedge$
$\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)) \vee$
$\neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau)$

Reformulating the property to emphasize the form of the sets over their content, plus some math to remove redundant terms, we get the following:

$\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle = M \wedge M \neq \{\} \wedge \text{state}(P,\sigma)$
$\mapsto$
$(\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ Q,\mu' : \text{message}(Q,P,\tau,\mu') :: \mu'\ \rangle \subset M) \vee$
$(\neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau))$

Since F2 requires that there is always exactly one σ for which *state(P,σ)* is true, and since σ does not appear anywhere else in the property, we can apply the general disjunction rule for **leads-to** to remove *state(P,σ)* from the left-hand-side, giving:

$\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle = M \wedge M \neq \{\}$
$\mapsto$
$(\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ Q,\mu' : \text{message}(Q,P,\tau,\mu') :: \mu'\ \rangle \subset M) \vee$
$(\neg\text{message}(P,Q,\tau,\mu) \wedge \text{gclock}(\tau))$

Finally, observe that

$\text{message}(P,Q,\tau,\mu) \Rightarrow \langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle \neq \{\}$

which allows us to remove the $M \neq \{\}$ disjunct from the lhs, giving

$$\begin{array}{l} \quad \text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ Q',\mu' : \text{message}(Q',P,\tau,\mu') :: \mu'\ \rangle = M \\ \mapsto \\ \quad (\text{gclock}(\tau) \wedge \text{message}(P,Q,\tau,\mu) \wedge \langle\ \textbf{set}\ P',Q',\mu' : \text{message}(P',Q',\tau,\mu') :: \mu'\ \rangle \subset M) \vee \\ \quad (\text{gclock}(\tau) \wedge \neg\text{message}(P,Q,\tau,\mu)) \end{array}$$

Which implies L4. □

Note. We can also use L4 to prove another result:

L5: $\quad \text{gclock}(\tau) \mapsto \langle\ \forall\ P,Q,\mu :: \neg\text{message}(P,Q,\tau,\mu)\ \rangle$

L5 implies F19.2 (by application PSP over L5 and F14), which can therefore be dropped from the specification. The proof of L5 also uses the induction principle for leads-to, this time over the size of the set of messages, as opposed to its content. From L4, we can conclude the following:

$$\begin{array}{ll} \text{L6:} & \quad \text{gclock}(\tau) \wedge \langle\ \Sigma\ P,Q,\mu : \text{message}(P,Q,\tau,\mu) :: 1\ \rangle = M \wedge M > 0 \\ & \mapsto \\ & \quad \text{gclock}(\tau) \wedge \langle\ \Sigma\ P,Q,\mu : \text{message}(P,Q,\tau,\mu) :: 1\ \rangle < M \end{array}$$

using the general disjunction theorem for **leads-to.** To complete the proof that L4 implies L5, we must be able to show the following:

$$\begin{array}{ll} \text{L7:} & \quad \text{gclock}(\tau) \wedge \langle\ \Sigma\ P,Q,\mu : \text{message}(P,Q,\tau,\mu :: 1\ \rangle = M \\ & \mapsto \\ & \quad (\text{gclock}(\tau) \wedge \langle\ \Sigma\ P,Q,\mu : \text{message}(P,Q,\tau,\mu) :: 1\ \rangle < M) \vee \\ & \quad \langle\ \forall\ P,Q,\mu :: \neg\text{message}(P,Q,\tau,\mu)\ \rangle \end{array}$$

The result follows immediately from L6 and

$$\text{gclock}(\tau) \wedge \langle\ \Sigma\ P,Q,\mu : \text{message}(P,Q,\tau,\mu) :: 1\ \rangle = 0 \Rightarrow \text{gclock}(\tau) \wedge \langle\ \forall\ P,Q,\mu :: \neg\text{message}(P,Q,\tau,\mu)\ \rangle$$ □

Refined Specification.

Safety: F1-F3, F5, F7-F17, F21

Progress: F18.2.1, F19.3, F20

5.6. Add upper bound on time increment

We are now in a position to perform the second refinement suggested in section 4.3. While F20 provides a lower bound on the time increment (namely 1), it does not include the upper bound. Such an upper bound is provided by F21, since the time may never exceed the earliest message or action time. By folding F21 into F20, the progress property expresses all the constraints on the clock movement, which is convenient when the time comes to write the abstract program.

Refinement. F20 is replaced by F20.1, which requires that the clock move forward whenever the earliest work to be done is in the future.

F20.1
$$\begin{array}{l} \text{gclock}(T) \wedge \\ (\text{message}(P,Q,\tau',\mu) \vee (\text{action}(P,\tau',\alpha) \wedge \alpha \neq \perp)) \wedge \\ \tau = \langle\ \textbf{min}\ P,Q,T',\mu,\alpha : \text{message}(P,Q,T',\mu) \vee (\text{action}(P,T',\alpha) \wedge \alpha \neq \perp)) :: T' \rangle \wedge \\ \tau > T \\ \mapsto \\ \langle\ \exists\ T' : \tau \geq T' \geq T+1 :: \text{gclock}(T') \rangle \end{array}$$

Proof Obligations.

F20.1 $\vdash$ F20.

Proof Outline. From L5 (and an analogous property over actions, the proof of which is omitted), we can see eventually the system reaches a state in which there are no messages or actions for the current time. That is, we have the property:

L8: $\text{gclock}(T) \mapsto \langle\ \forall\ P,Q,\mu :: \neg\text{message}(P,Q,T,\mu) \rangle \wedge \langle\ \forall\ P,\alpha :: \neg\text{action}(P,T,\alpha) \vee \alpha = \perp \rangle$

F21 thus implies that the minimum activity is in the future, and from F20.1, the clock must move forward. Since The left-hand side of F20 implies the left-had side of L8, we have F20.

Refined Specification.

Safety: F1-F3, F5, F7-F17, F21

Progress: F18.2.1, F19.3, F20.1

5.7. The Final Specification.

At this point, the three progress properties describe transformations which can easily be considered atomic, which was our goal in the refinement process. The complete specification is reproduced below.

F1: **inv.** $\langle \Sigma\ T : gclock(T) :: 1 \rangle = 1$

F2: **inv.** $\langle \Sigma\ \sigma : state(P,\sigma) :: 1 \rangle = 1$

F3: **inv.** $\langle \Sigma\ \tau,\alpha : action(P,\tau,\alpha) :: 1 \rangle = 1$

F5: **inv.** $message(P,Q,\tau,\mu) \wedge message(P',Q',\tau',\mu') \wedge (P,Q,\tau) \neq (P',Q',\tau') \Rightarrow \mu \neq \mu'$

F7: $gclock(\tau)$ **unless** $\langle \exists\ \tau' : \tau' \geq \tau+1 :: gclock(\tau') \rangle$

F8: $action(P,\tau,\alpha) \wedge \langle$ **set** $Q,\tau',\mu : message(P,Q,\tau',\mu) :: \mu \rangle = M$
unless
$\neg action(P,\tau,\alpha) \vee \langle$ **set** $Q,\tau',\mu : message(P,Q,\tau',\mu) :: \mu \rangle \subset M$

F9: $message(P,Q,\tau,\mu)$ **unless** $message(P,Q,\tau,\mu) \wedge gclock(\tau)$

F10: $message(P,Q,\tau,\mu) \wedge gclock(\tau)$ **unless** $\neg message(P,Q,\tau,\mu) \wedge gclock(\tau)$

F11: $\neg message(P,Q,\tau,\mu) \wedge gclock(\tau)$ **unless** $\neg gclock(\tau)$

F12: $state(P,\sigma) \wedge gclock(\tau) \wedge \langle$ **set** $Q,\mu : message(Q,P,\tau,\mu) :: \mu \rangle = M \wedge M \neq \{\}$
unless
$\langle \exists\ \mu : \langle$ **set** $Q,\mu' : message(Q,P,\tau,\mu') :: \mu' \rangle = M-\{\mu\} :: state(P,u(P,\sigma,\mu)) \rangle$

F13: $action(P,\tau,\alpha) \wedge \alpha \neq \perp$ **unless** $action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau)$

F14: $action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau)$
unless
$action(P,\tau,\alpha) \wedge \alpha \neq \perp \wedge gclock(\tau) \wedge \langle \forall\ Q,\mu :: \neg message(Q,P,\tau,\mu) \rangle$

F15: $state(P,\sigma) \wedge enabled(P,\tau,\alpha)$
unless
$action(P,e(P,\sigma,\alpha),a(P,\sigma,\alpha)) \wedge$
$state(P,s(P,\sigma,\alpha)) \wedge$
$\langle \forall\ Q : c(P,Q,\sigma,\alpha) :: message(P,Q,l(P,Q,\tau,\sigma,\alpha),v(P,Q,\sigma,\alpha)) \rangle$

F16: **stable** $halt(P)$

F17: $state(P,\sigma) \wedge gclock(\tau)$
unless
$state(P,\sigma) \wedge \langle \exists\ Q,\tau',\alpha,\mu : gclock(\tau') :: enabled(P,\tau',\alpha) \vee message(Q,P,\tau',\mu) \rangle$

F18.2.1: $gclock(\tau) \wedge \langle$ **set** $Q,\mu : message(Q,P,\tau,\mu) :: \mu \rangle = M \wedge M \neq \{\} \wedge state(P,\sigma)$
$\mapsto$
$\langle \exists\ \mu : \langle$ **set** $Q,\mu' : message(Q,P,\tau,\mu') :: \mu' \rangle = M - \{\ \mu\ \} :: state(P,u(P,\sigma,\mu)) \rangle$

F19.3: $\text{enabled}(P,\tau,\alpha) \wedge \text{state}(P,\sigma)$
$\mapsto$
$\text{action}(P,e(P,\sigma,\alpha),a(P,\sigma,\alpha)) \wedge$
$\text{state}(P,s(P,\sigma,\alpha)) \wedge$
$\langle\, \forall\, Q : c(P,Q,\sigma,\alpha) :: \text{message}(P,Q,l(P,Q,\tau,\sigma,\alpha),v(P,Q,\sigma,\alpha)) \,\rangle$

F20.1 $\text{gclock}(T) \wedge$
$(\text{message}(P,Q,\tau',\mu) \vee (\text{action}(P,\tau',\alpha) \wedge \alpha \neq \perp)) \wedge$
$\tau = \langle\, \textbf{min}\ P,Q,T',\mu,\alpha : \text{message}(P,Q,T',\mu) \vee (\text{action}(P,T',\alpha) \wedge \alpha \neq \perp)) :: T' \,\rangle \wedge$
$\tau > T$
$\mapsto$
$\langle\, \exists\, T' : \tau \geq T' \geq T+1 :: \text{gclock}(T') \,\rangle$

F21: **inv.**
$((\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp) \vee \text{message}(P,Q,\tau,\mu)) \wedge \text{gclock}(T)$
$\Rightarrow$
$T \leq \langle\, \textbf{min}\ P,Q',\tau',\alpha',\mu' : (\text{action}(P,\tau,\alpha) \wedge \alpha \neq \perp) \vee \text{message}(P,Q,\tau,\mu) :: \tau \,\rangle$

5.8. Abstract Program

The three progress properties (F18.2.1, F19.3, and F20.1) suggest an abstract program having three transaction types: one to execute actions, one to move the clock, and one to deliver messages. The remainder of this section gives these transactions. Informally, each transaction consists of two subtransactions, one to establish the required progress condition, and one to continue the computation. The transactions are given without proof. Accompanying each transaction is a list of the properties which directly constrain its form, and which are therefore likely to be affected directly by any refinement of the transaction. We make use of these *characteristic properties* in the program refinement stages as a heuristic for reducing the amount of formal proof required. Note that this does not imply that it is not necessary to prove that the transaction satisfies the property, but rather that the proof can be deferred until the program refinement process is completed, since the refinements are unlikely to invalidate the property.

Property F18.2.1 requires that a messages with a delivery time equal to the current simulation time be incorporated into the state of the destination processor. We implement this as a global transaction of type *gdeliver*, which does this work for all messages at all processor.

gdeliver ≡
P,Q,τ,μ,σ :
gclock(τ), state(P,σ), message(Q,P,τ,μ) → message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
‖ **TRUE** → gdeliver

The characteristic properties for this transaction are F2, F9, F10, and F12.

Property F19.3 requires that the execution of an enabled action result in the atomic update of the processor's local state, the creation of the next action, and (possibly) the sending of messages to other processors. To accomplish this, we introduce a transaction type *task*, having the same form as *action*. with the obvious relationship:

F22: **inv.** [task(P,τ,α)] ⇔ action(P,τ,α)

Because this transaction affects the entire abstract state, it is necessary to verify that it violates none of the safety properties of the specification. A brief perusal of the specification identifies F2, F3, F13, F14, F15, and F16 as the characteristic set for this transaction.

task(P,τ,α) ≡
  σ :
    α ≠ ⊥, gclock(τ), ⟨ ∀ Q,μ :: ¬message(Q,P,τ,μ) ⟩, state(P,σ)
  →
    task(P,e(P,σ,α),a(P,σ,α)),
    state(P,σ)†, state(P,s(P,σ,α)),
    ⟨ Q : c(P,Q,σ,α) :: message(P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
  ‖ **NOR** → task(P,τ,α)

Finally, property F20.1 requires that time move forward when all the work to do is in the future. This is accomplished by introducing a transaction of type *gtick* which examines the global system state and moves the clock forward when there are no *tasks* or *messages* to be processed at the current time.

gtick ≡
  T,T' :
    gclock(T),
    ⟨ ∀ P,Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ > T ⟩,
    T+1 ≤ T' ≤ ⟨ **min** P,Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ ⟩
  →
    gclock(T)†, gclock(T')
  ‖ **TRUE** → gtick

As with the other transactions, there is no need to prove that this transaction satisfies most of the safety properties. In particular, we need only provide proofs for properties F1, F7, F10, F11, F14, F15, and F21.

6. Architecture-Directed Program Refinement

The abstract program definitely solves the simulation problem. However, the widespread use of global data items makes it unsuitable for implementation on any truly distributed architecture. Our refinement methodology now turns to address this problem. In the next two sections, we give formal descriptions of the constraints imposed

on programs by two example architectures, and then use these constraints to derive a solution specifically tailored to each architecture.

6.1. Consensus Bus

Our first example architecture is a classic distributed memory message-passing architecture with a special-purpose device which makes it particularly well-suited for use in distributed simulation. The target machine consists of a number of identical components called *sites*. Each site h contains a controller k, a processor e, and two memory units, r (registers) and m (main memory). We assume that all entities in the physical system are assigned distinguishable identifiers which we can use to refer to each component, and let H denote the set of site identifiers. The sites are connected by a data bus D and a consensus bus C. Both the controller and the processor are sequential machines capable of executing one operation at a time. At each site, the two processing units run asynchronously; that is, they do not share a common clock. The registers are used to allow the controller and processor at a single site to share a limited amount of data. Reads and writes to the registers are atomic. Figure 2 shows a stylized representation of a single site.

The data bus is the primary mechanism for sharing information between sites. This bus connects the processors and main memories from all sites, creating a limited form of distributed shared memory. Using the data bus, processors can write to, but not read from, memories at other sites. In a sense, the data bus implements a message-passing channel which allows processors to pass data amongst themselves by leaving messages in mailboxes. The local memories can be read only by the processor at the same site, no other memory accesses are permitted. In particular, the controllers cannot access the main memories, and the register units can only be accessed by the processing units at the same site. All accesses to the main memories (both reads and writes) are atomic.

The consensus bus is a specialized hardware device which allows all the controller elements to share a limited amount of information in the following manner. Logically, during the execution of a single step, the controllers at each site provide to the bus a data value. The consensus bus computes a hardwired function of these values, the result of which is then provided to the controllers and can be used in the remainder of the step. The result is a synchronous, lock-step execution by the controllers.

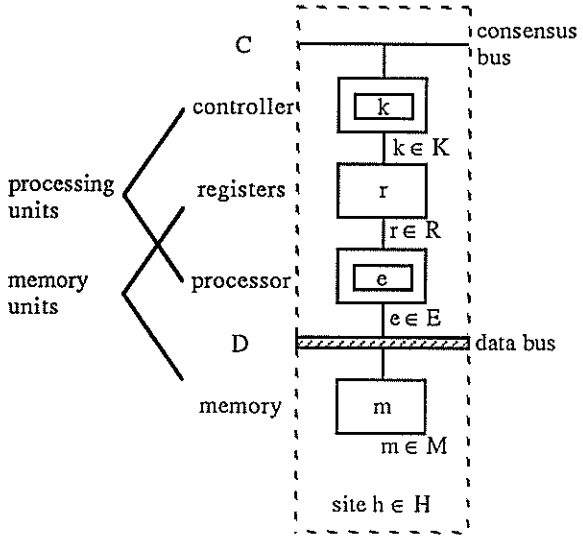

Figure 2: One site of the postulated architecture.

Allocation Constraints. Now, we can define formally a *site h* as a five-tuple consisting of the site identifier *h.id* and the identifiers for the individual hardware components associated with the site: a controller *h.k*, a bank of registers *h.r*, a processor *h.e*, and a memory *h.m*. *H* is the set consisting of all site identifiers. We also introduce the set *K* of controller identifiers, the set *R* of register identifiers, the set *E* of processor identifiers, and the set *M* of memory identifiers. Since each hardware component belongs to one and only one site we find it convenient to introduce a function:

$$site : K \cup R \cup E \cup M \rightarrow H$$

which given the identifier of a component, returns the identifier for the site to which the component belongs.

We now consider formally the constraints on allocation of programs and data to sites. For this purpose, we can introduce a predicate *locus(i,j)*, where *i* is from the set of transactions or the set of tuples, and *j* is from the set of identifiers for hardware components. The predicate *locus(i,j)* is *true* if the dataspace element *i* is allocated to component *j*, and is initially restricted so that transactions are mapped to processing units, and data tuples to memory units:

$$t \in \mathbf{TRS} \wedge \text{locus}(t,i) \Rightarrow i \in E \cup K$$

$v \in \mathbf{TPS} \wedge \text{locus}(v,i) \Rightarrow i \in R \cup M$

This very general definition of the locus predicate can be further constrained to reflect the specifics of the simulator hardware. In the present case, the sequential nature of the processing units constrains the allocation of software to hardware, allowing at most one transaction to be present at any time on each of the processing units:

C_C1: **inv.** $[t_1] \wedge [t_2] \wedge t_1 \neq t_2 \wedge \text{locus}(t_1,i_1) \wedge \text{locus}(t_2,i_2) \Rightarrow i_1 \neq i_2$

(The notation $[t]$ means "the transaction instance t exists in the dataspace.") Also, because the processors run asynchronously with respect to the rest of the system, transactions which are placed on them cannot be part of any synchronic group containing transactions allocated to other processing units:

C_C2: **inv.** $[t_1] \wedge [t_2] \wedge t_1 \neq t_2 \wedge \text{locus}(t_1,i) \wedge i \in E \Rightarrow \neg (t_1 \approx t_2)$

Finally, since controllers execute synchronously, transactions which are placed on them must all be part of the same synchronic group.

C_C3: **inv.** $[t_1] \wedge [t_2] \wedge \text{locus}(t_1,i_1) \wedge i_1 \in K \wedge \text{locus}(t_2,i_2) \wedge i_2 \in K \Rightarrow t_1 \approx t_2$

Access Restrictions. We turn now to the constraints on memory access by transactions. In particular, we wish to constrain the locations of reads and writes made by transactions. Our approach is to introduce auxiliary tuples which record the reading or writing of a variable by a program running on one of the processing units. We can describe the read/write constraints in terms of invariants over two auxiliary tuples, *raccess(i,j)* and *waccess(i,j)*, where $i \in K \cup E$, and $j \in K \cup R \cup E \cup M$ (we include the possibility that a transaction is read or written). The presence of one of these tuples in the dataspace indicates that a transaction with locus i has read (or written) an entity with locus j. To prove that a transaction satisfies one of these constraints, all subtransactions are augmented to insert an *raccess* tuple whenever a tuple appears in the query of the subtransaction, and to insert a *waccess* tuple whenever a tuple appears in the action. The access constraints can now be expressed in terms of three invariants.

C_C4: *Transactions on the processors can only read from memories located at the same site*
inv. $i \in E \wedge \text{raccess}(i,j) \Rightarrow j \in R \cup M \wedge \text{site}(i) = \text{site}(j)$

C_C5: *Transactions on processors can write to any memory, to registers at the same site, and to the processor itself (to change execution state)*
inv. $i \in E \wedge \text{waccess}(i,j) \Rightarrow j \in M \vee (j \in R \cup E \wedge \text{site}(i) = \text{site}(j))$

C_C6: *Transactions on controllers can read and write only registers at the same site or the controller itself*
inv. $i \in K \wedge (raccess(i,j) \vee waccess(i,j)) \Rightarrow j \in K \cup R \wedge site(i) = site(j)$

Consensus bus. The consensus bus is actually a specialized device which synchronizes the execution of all controllers. Additionally, at each step the bus accepts a boolean value from each controller and returns to all controllers the result of applying the logical *and* across all the boolean values received. There are two reasons for introducing the consensus bus in our illustration. First, from a practical viewpoint, such a bus is easy to construct and matches the needs of the simulator. Second, from a pedagogical perspective, the bus allows us to illustrate the formalization of a highly specialized device and the expressive power of the synchronic group construct in Swarm.

Returning to the formalization of the constraints imposed by the consensus bus, we take advantage of the built-in consensus feature associated with synchronic groups. It allows us to reduce the effect of the consensus bus to restricting transactions allocated to the controllers from using any of the special queries except for **AND** and **NAND** (and of course, **TRUE**). To accomplish this, we augment each subtransaction on controller k that uses **OR** (or **NOR**), e.g.,

|| : **OR**, *query* → *action*

so as to insert an auxiliary tuple that records the improper use of the consensus bus, i.e.,

|| : **OR**, *query* → *action*, invalid_consensus()

and add the following proof obligation:

C_C7: **inv.** ¬ invalid_consensus()

We rely here upon C_C3, which requires all transactions allocated to controllers to execute synchronously, allowing us to make use of the Swarm consensus mechanism. This formulation is in fact stronger than a syntactic restriction; for example, if the query associated with the **OR** always evaluates to *false*, we can prove that no improper use of the consensus bus takes place in spite of the fact that the syntax alone suggests otherwise.

The formalization of architectural constraints clearly depends upon the computational model being used. The fact that Swarm already provides a form of built-in consensus makes the formalization trivial. This does not mean that our assertional method would be inappropriate otherwise, but it does mean that the formalization, by ne-

cessity, would be more complex. In the absence of the built-in consensus, auxiliary tuples would be needed to keep track of the booleans supplied by each controller and the values returned by the bus. The relation between these values would, of course, be constrained by an appropriate invariant.

Initial Mapping. We now return to the task of mapping the abstract program onto this architecture. Obviously, not all of the abstract program's state can be allocated without violating the architectural constraints. This fact will drive the derivation process as we move to resolve those allocation decisions which cannot be made at this point. Nevertheless, certain allocations are required by the nature of the architecture; for example, the *task* transaction must be allocated to the processors as the *tasks* at each processor must run asynchronously, the processor's *state* should be placed into the processor's local memory, and *messages* which must be sent from one processor to another, should also be allocated to memories.

A_C1: **inv.** locus(task(P,τ,α),i) ⇒ i ∈ E
A_C2: **inv.** locus(state(P,σ),i) ⇒ i ∈ M
A_C3: **inv.** locus(message(P,Q,τ,μ),i) ⇒ i ∈ M
A_C4: **inv.** locus(state(P,σ),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ site(i_1) = site(i_2)
A_C5: **inv.** locus(message(Q,P,τ,μ),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ site(i_1) = site(i_2)

Note that this allocation satisfies C_C1-C_C2 for the entire abstract state, and that C_C3, C_C6 and C_C7 are satisfied vacuously (since no transaction is allocated to the controllers). However, there is no allocation of either *gdeliver* or *gtick* that satisfies the requirements. Obviously, neither can be placed on the processors without violating C_C1, and allocating them to the controllers would violate C_C6, since each needs access to information located in the site's main memory. Further, the *gclock* tuple cannot be allocated to any memory or register bank, as it must be read by every processor, violating either C_C4 or C_C6. The remainder of the refinement process is guided by these failures.

Allocate *gdeliver*. The problem of resolving the allocation of *gdeliver* is most easily solved, so we address it first. Since the work done by the transaction is clearly local, the obvious solution is to distribute the processing to each site, introducing a local *deliver* transaction, with one copy of the transaction for each node.

Refinement. The *gdeliver* transaction type is replaced by a *deliver* transaction type, formed by parameterizing *gdeliver* with the node id, as follows:

deliver(P) ≡
    Q,τ,μ,σ :
        gclock(τ), state(P,σ), message(Q,P,τ,μ) → message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
    ‖ **TRUE** → deliver(P)

We require that there be a *deliver* transaction for each processor, i.e.,

C_C8: **inv.** ⟨ ∃ Q,τ,α,μ :: (action(Q,τ,α) ∧ α ≠ ⊥) ∨ message(Q,P,τ,μ) ⟩ ⇒ deliver(P)[4]

This formulation requires that a *deliver* transaction exists if there is the possibility of any new messages appearing in the future, while allowing the program to terminate when the simulation is complete. Additionally, we introduce further refinements on *locus* to guarantee that the transaction satisfies the access restrictions (C_C4 and C_C5):

A_C6: **inv.** locus(deliver(P),i) ⇒ i ∈ E

A_C7: **inv.** locus(deliver(P),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ site(i_1) = site(i_2)

Re-Verification Obligations. All the safety properties associated with *gdeliver* (F2, F9, F10, F12, or F18.2.1) continue to be satisfied, since no new state transitions are introduced by this refinement. The progress property (F18.2.1) also continues to hold. This can be seen by observing that any transition which would have been made by the global transaction will be performed by one of the local transactions (specifically the one assigned to the destination site); and since the local transactions are always re-created, the statement needed to make the transition always exists.

Outstanding Violations. This allocation violates C_C1 as *task* is already allocated to the processor, but we will deal with this violation later. Additionally, neither *gclock* nor *gtick* can yet be allocated.

Allocate *gclock(T)*. We now turn to the problem of allocating the clock. Since *task* reads *gclock*, C_C4 applies, requiring that *gclock* be located on every processor which has a *task*, and that the tuple be located in either registers or memory. This suggests that a local clock should be maintained at each site, with all clocks running in lockstep.

Refinement. We introduce a tuple of type *clock*, with one tuple for each node. The tuple replaces the *gclock* tuple type. All three transactions must be re-written to make use of the new data representation.

task(P,τ,α) ≡
    σ :
        α ≠ ⊥, clock(P,τ), ⟨ ∀ Q,μ :: ¬message(Q,P,τ,μ) ⟩, state(P,σ)
    →
        task(P,e(P,σ,α),a(P,σ,α)),
        state(P,σ)†, state(P,s(P,σ,α)),
        ⟨ Q : c(P,Q,σ,α) :: message(P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
    ‖ **NOR** → task(P,τ,α)

gtick ≡
    P',T,T' :
        clock(P',T),
        ⟨ ∀ P,Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ > T ⟩,
        T+1 ≤ T' ≤ ⟨ **min** P,Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ ⟩
    →
        ⟨ P : P ∈ *nodes* :: clock(P,T)†, clock(P,T') ⟩
    ‖ **TRUE** → gtick

deliver(P) ≡
    Q,τ,μ,σ :
        clock(P,τ), state(P,σ), message(Q,P,τ,μ) → message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
    ‖ **TRUE** → deliver(P)

We introduce the obvious requirements that there be exactly one clock for each processor, and that all clocks carry the same time.

C_C9: **inv.** ⟨ Σ T : clock(P,T) :: 1 ⟩ = 1
C_C10: **inv.** gclock(T) ≡ ⟨ ∀ P :: clock(P,T) ⟩

Additionally, to resolve the violation of C_C4 by *task* and *deliver*, we must allocate the *clock* tuple to either registers or memories at the same site as the two transactions. That is, we introduce the following additional restrictions on *locus* :

A_C8: **inv.** locus(clock(P,T),i) ⇒ i ∈ M ∪ R
A_C9: **inv.** locus(clock(P,T),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ site(i_1) = site(i_2)

Re-Verification Obligations. The transactions *task* and *deliver* are actually unaffected by this transformation since *gclock(T)* ≡ *clock(P,T)*. As far as *gtick* is concerned, it clearly satisfies C_C10, and consequently, all other behavioral obligations. Additionally, we can now show that the *task* and *deliver* transactions access only local data.

Outstanding Violations. Both *task* and *deliver* now satisfy all architectural constraints except for C_C1. *gtick* cannot be allocated without violating one of C_C4-C_C6.

Allocate *gtick*. As we turn to address the problem of allocating *gtick*, note that the "∀" in the query suggests using the consensus bus, so an allocation to the controllers is proposed. With this in mind, we introduce a distributed processing scheme, with one *tick* transaction at each site.

Refinement. The refinement is presented in two steps. First, we present a formulation with a separate subtransaction for each processor *P*. Each subtransaction checks locally for work to be performed. The global check is then done using an **AND** special predicate query, which succeeds only when each of the local subtransactions succeed (that is, all the work at each site is in the future). Additionally, we must select a time increment that does not violate F21. Since we cannot pass any additional information between controllers, and since all sites must have the same time value, 1 is the only reasonable choice.

gtick ≡
⟨ ‖ P : P ∈ *nodes* ::
T : clock(P,T), ⟨ ∀ Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ > T ⟩ → **skip**
‖ T : **AND**, clock(P,T) → clock(P,T)†, clock(P,T+1)
⟩
‖ **TRUE** → gtick

Now we can separate the *gtick* transaction into a collection of local, synchronous *tick* transactions, one for each processor, with the **AND** special query computed by the consensus bus. The parameter to the local transaction is the *P* from the subtransaction generator. This transformation retains the semantics of the previous transaction, since the individual *tick* transactions are in the same synchronic group. This gives us the following definition for the new transaction:

tick(P) ≡
T : clock(P,T), ⟨ ∀ Q,τ,α,μ : message(Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ > T ⟩ → **skip**
‖ T : **AND**, clock(P,T) → clock(P,T)†, clock(P,T+1)
‖ **TRUE** → tick(P)

To maintain the invariant that all clocks have the same value, we must require that, if any *tick* transaction remains in the dataspace, then all transactions remain, i.e.,

C_C11: **inv.** tick(P) ⇔ tick(Q)

Since we make use of the consensus bus, we will need to allocate the transactions to the controllers:

A_C10: **inv.** locus(tick(P),i) ⇒ i ∈ K
A_C11: **inv.** locus(tick(P),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ site(i_1) = site(i_2)
C_C12: **inv.** tick(P)~tick(Q)

Finally, since the *tick* transaction reads and writes the *clock* tuple, that tuple must be allocated to the registers, allowing us to refine the *locus* of *clock* further:

A_C12: **inv.** locus(clock(P,T),i) $\Rightarrow$ i $\in$ R

Re-Verification Obligations. Because the combination of *tick* transactions is functionally equivalent to the original *gtick*, no re-verification is required.

Outstanding Violations. We are left with a violation of C_C6 by *tick*, since it must determine the existence or non-existence of messages and tasks, neither of which are allocated to the registers. The violations of C_C1 by *task* and *deliver* are also outstanding.

Detecting absence of work. To remove the violation of C_C6 by *tick*, we introduce two new tuple types, allocated to the registers, which contain sufficient information to allow *tick* to detect that there is no remaining work to be done at a site at the current time.

Refinement. We introduce the tuple types *event(P,τ)* and *no_msg(P)*, having the meanings "the next *task* to execute for node *P* is at time τ," and "there are no remaining unabsorbed *messages* for node *P* at the present time," respectively.

A_C13: **inv.** locus(event(P,τ),i) $\Rightarrow$ i $\in$ R
A_C14: **inv.** locus(event(P,τ),i_1) $\wedge$ locus(task(P,τ,α),i_2) $\Rightarrow$ site(i_1) = site(i_2)
A_C15: **inv.** locus(no_msg(P),i) $\Rightarrow$ i $\in$ R
A_C16: **inv.** locus(no_msg(P),i_1) $\wedge$ locus(task(P),i_2) $\Rightarrow$ site(i_1) = site(i_2)

The *task* transaction maintains the *event* tuple, while *deliver* and *tick* cooperate to update *no_msg*.

```
task(P,τ,α) ≡
    σ :
        α ≠ ⊥, clock(P,τ), no_msg(P), state(P,σ)
    →
        event(P,τ)†,
        task(P,e(P,σ,α),a(P,σ,α)), ⟨ : a(P,σ,α) ≠ ⊥ :: event(P,e(P,σ,α)) ⟩,
        state(P,σ)†, state(P,s(P,σ,α)),
        ⟨ Q : c(P,Q,σ,α) :: message(P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
    ||  NOR → task(P,τ,α)

tick(P) ≡
        T : clock(P,T), no_msg(P), ¬event(P,T) → skip
    ||  T : AND, clock(P,T) → clock(P,T)†, clock(P,T+1), no_msg(P)†
    ||  TRUE→ tick(P)
```

deliver(P) ≡
    Q,τ,μ,σ :
        clock(P,τ), state(P,σ), message(Q,P,τ,μ) → message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
    ‖ τ : clock(P,τ), ⟨ ∀ Q,μ :: ¬message(Q,P,τ,μ) ⟩ → no_msg(P)
    ‖ **TRUE** → deliver(P)

Re-Verification Obligations. We must show that the new forms of each transaction satisfy the corresponding characteristic properties. We can also show that there are now no access violations by *tick*.

Proof Outline. To prove the functional correctness of this refinement, we must show that the new representation carries the same semantics as the original. In particular, we must show that *tick* cannot incorrectly detect the absence of work. This can be accomplished if the refined transactions maintain the following invariants, which equate the two data representations:

C_C13: **inv.** event(P,τ) ≡ ⟨ ∃ α : task(P,τ,α) :: α ≠ ⊥ ⟩
C_C14: **inv.** no_msg(P) ∧ clock(P,τ) ⇒ ⟨ ∀ Q,μ :: ¬message(Q,P,τ,μ) ⟩

The truth of these invariants is clear from the text of the transactions. Since the *no_msg* tuple is removed by *tick* when the clock is advanced (to maintain C_C14), we require a progress property that guarantees that the removal of the last message for a node results in the re-insertion of *no_msg*. That is, we require:

C_C15: no_msg(P) **detects** clock(P,τ) ∧ ⟨ ∀ Q,μ :: ¬message(Q,P,τ,μ) ⟩

This property follows immediately from the invariance of C_C14 and the text of *deliver*.

Outstanding Violations. Only the violation of C_C1 by *task* and *deliver* is outstanding.

Satisfy uniprogramming requirements for processors. Since the *deliver* and *task* transactions require access to both the registers and memory at a site, both must be allocated to the processors. However, at most one transaction can be present on any processor. To satisfy C_C1, we must combine these 2 transactions in some way. This can be done either by combining the two transactions into a single transaction that does both, or by alternating them. We opt for the latter, since this more closely reflects the approach that might be used in a traditional programming language.

Refinement. We modify *task* and *deliver* as follows:

task(P,τ,α) ≡
  σ :
    α ≠ ⊥, clock(P,τ), no_msg(P), state(P,σ)
  →
    event(P,τ)†, deliver(P,a(P,σ,α)), ⟨ : a(P,σ,α) ≠ ⊥ :: event(P,e(P,σ,α)) ⟩,
    state(P,σ)†, state(P,s(P,σ,α)),
    ⟨ Q : c(P,Q,σ,α) :: message(P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
  || **NOR** → task(P,τ,α)

deliver(P,α) ≡
  Q,τ,μ,σ :
    clock(P,τ), state(P,σ), message(Q,P,τ,μ)
  →
    message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ)), deliver(P,α)
  || **NOR** → no_msg(P)
  || τ : **NOR**, clock(P,τ), event(P,τ) → task(P,τ,α)
  || T,τ : **NOR**, clock(P,τ), ((event(P,T) ∧ T ≠ τ) ∨ α = ⊥) → deliver(P,α)

with the additional requirement that only one of the two transactions is present at any given time:
  C_C16: **inv.** ¬(task(P,τ,α) ∧ deliver(P,α'))

We have added a parameter to *deliver* to keep track of the action which should be performed when all messages have been delivered. There is no need to include the time of the next action, since that is already available from the *event* tuple.

Proof Obligations. This transformation does not affect any of the safety properties to be satisfied by *task* and *deliver*. We must, however, show that the transactions satisfy F18.2.1, the progress property which was previously satisfied by *deliver*, and F19.3, the progress property for *task*. Finally, it is obvious that the new transactions satisfy C_C1 and, therefore, all architectural constraints are met at this point.

Proof Outline. To prove that messages are eventually delivered (F18.2.1), we must show that if there is a message to be delivered at the current time, then there is a *deliver* transaction to process it, that is

inv. clock(P,τ) ∧ message(Q,P,τ,μ) ⇒ ⟨ ∃ α :: deliver(P,α) ⟩

This invariant can be verified by examining the program text: *deliver* does not move the time nor create messages; *tick* does not create messages, and only advances the time when there are no messages and no *task*; and once a *task* is

executed, it creates the next *deliver* transaction, with all messages being in the future. Since the new form of *deliver* clearly performs the necessary state transitions, then we have F18.2.1.

Additionally, we must show that if there is an event scheduled at any given time, there will eventually be a *task* transaction to perform it, e.g.,

event(P,τ) **until** task(P,τ,α)

The **unless** is clearly maintained by the entire program, and the **leads-to** is established by the third subtransaction of *deliver*.

This completes the derivation of the simulation program for execution on the consensus bus architecture. To simplify the presentation, we did not consider the problem of termination, although it should be clear that the issue could have been addressed at the expense of slightly more complex formulations of some properties. We now turn our consideration to solving the problem on a very different architecture, to show the flexibility of this approach.

6.2. Ring

Our second example architecture is a traditional ring. The nodes in the ring are multiprogrammable, general purpose processors containing a local memory. Each processor is assumed to have a unique identifier from the set { 0 .. N-1 }, where *N* is the size of the ring. Identifiers are assigned sequentially around the ring. The nodes on the ring are connected by one-way, asynchronous communication channels (communication is clockwise around the ring). This is a true distributed memory architecture, so a process can only directly read from its local memory. We model message passing as writes by one processor to another processor's memory; because communication proceeds clockwise around the ring, a processor can write both to its local memory, and to the memory of the process to its immediate right in the ring. We assume that all memory accesses are atomic. To simplify discussions about the ring, we introduce two definitions.

right(P) $\equiv$ (P+1) mod N

R twixt (P,Q) $\equiv$ (P > Q $\wedge$ (P $\leq$ R < N $\vee$ 0 $\leq$ R $\leq$ Q)) $\vee$ (P $\leq$ Q $\wedge$ P $\leq$ R $\leq$ Q)

Allocation Constraints. As with the consensus bus, we will express the mapping of program elements to hardware using the predicate *locus(i,j)*, having the meaning that the transaction or tuple *i* is to be allocated to the processor with identifier *j*.

Access Restrictions. We introduce two auxiliary tuple types *raccess(i,j)* and *waccess(i,j)* to record reads and writes. Both *i* and *j* are integers in the range 0..N-1, and have the meaning that a transaction resident on node *i* of the ring has read (or written) a tuple or transaction on node *j*. The informal access constraints can be formally expressed using two predicates over these auxiliary tuples.

C_R1: **inv.** raccess(i,j) $\Rightarrow$ i = j

C_R2: **inv.** waccess(i,j) $\Rightarrow$ i = j $\vee$ j = right(i)

Additional Restrictions. The only additional constraint introduced by the ring architecture is the requirement that the nodes execute asynchronously. This can be succinctly stated within the Swarm notation using the $\approx$ notation, as:

C_R3: **inv.** $[t_1] \wedge [t_2] \wedge \text{locus}(t_1,i_1) \wedge \text{locus}(t_2,i_2) \wedge i_1 \neq i_2 \Rightarrow \neg(t_1 \approx t_2)$

Initial Mapping. Very little of the abstract program's initial state can be allocated without further refinement. While we can be certain that we want each processor's local *state* to be allocated to the same location as the *task* transaction (A_R1), no other decisions are possible.

A_R1: **inv.** $\text{locus}(\text{task}(P,\tau,\alpha),i_1) \wedge \text{locus}(\text{state}(P,\sigma),i_2) \Rightarrow i_1 = i_2$

Several problems are immediately obvious. First, *gclock* cannot be allocated to any node in the ring, since its value is read by every *task*. Further, neither of the global transactions (*gdeliver* and *gtick*) can be placed until *gclock* is dealt with. Finally, the restriction on writes (C_R2) makes it impossible for a transaction to send a message to any transaction allocated to a processor other than the one to its right. Until the allocation decisions are made, it is not possible to prove that any of the transactions satisfy the access constraints (C_R1 and C_R2), since each transaction reads or writes the clock.

Allocate *gdeliver*. The problem of allocating *gdeliver* is the most easily solved. By C_R2, any transaction which updates a *state* tuple must be located on the same node as the *state* tuple. As in the consensus bus architecture, this implies a distributed version of *gdeliver*, with one transaction for each *P*. The refinement is identical to that in the previous example, so we give it without further comment.

deliver(P) ≡
  Q,τ,μ,σ :
    gclock(τ), state(P,σ), message(Q,P,τ,μ) → message(Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
  ‖ **TRUE** → deliver(P)

with

C_R4: **inv.** ⟨ ∃ Q,τ,α,μ :: (action(Q,τ,α) ∧ α ≠ ⊥) ∨ message(Q,P,τ,μ) ⟩ ⇒ deliver(P)
A_R2: **inv.** locus(deliver(P),i_1) ∧ locus(task(P,τ,α),i_2) ⇒ $i_1 = i_2$

Outstanding Violations. We still cannot allocate the *gclock* tuple, nor can we allocate the *gtick* transaction. The violation of the write restrictions (C_R2) with regards to messages also remains, and will be considered next.

Add a "current location" field to messages. Obviously, messages must be sent around the ring, since we cannot send messages directly between non-adjacent processors. As a first step, we will modify the form of messages to accommodate the routing process.

Refinement. We begin with a simple data refinement, refining the structure of messages by adding an additional processor identifier which gives the current location of the message, e.g.

msg(P,Q,R,τ,μ)

with the meaning, "*Q* has sent a message with content μ to *R*; it is currently at node *P* and must be delivered at time τ." Our coupling invariant (C_R5) states that a *message* exists if there is an analogous *msg* located somewhere between the source and destination processors.

C_R5: **inv.** message(Q,R,τ,μ) ≡ ⟨ ∃ P : P twixt(Q,R) :: msg(P,Q,R,τ,μ) ⟩

Additionally, we require that there be at most one *msg* tuple for any *message* (C_R6), and that it be allocated to the processor simulating node *P* (A_R3):

C_R6: **inv.** ⟨ Σ P : msg(P,Q,R,τ,μ) :: 1 ⟩ ≤ 1
A_R3: **inv.** locus(deliver(P),i_1) ∧ locus(msg(P,Q,R,τ,μ),i_2) ⇒ $i_1 = i_2$

We update the program to use the new representation. Note that the sending of a *msg* by *task* is performed by writing the tuple directly to the destination processor.

task(P,τ,α) ≡
    σ :
        α ≠ ⊥, gclock(τ), ⟨ ∀ Q,R,μ :: ¬msg(Q,R,P,τ,μ) ⟩, state(P,σ)
    →
        task(P,e(P,σ,α),a(P,σ,α)), state(P,σ)†, state(P,s(P,σ,α)),
        ⟨ Q : c(P,Q,σ,α) :: msg(Q,P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
    || **NOR** → task(P,τ,α)

deliver(P) ≡
    Q,τ,μ,σ :
        gclock(τ), state(P,σ), msg(P,Q,P,τ,μ) → msg(P,Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
    || **TRUE** → deliver(P)

gtick ≡
    T,T' :
        gclock(T),
        ⟨ ∀ P,Q,R,τ,α,μ : msg(R,Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ > T ⟩,
        T+1 ≤ T' ≤ ⟨ **min** P,Q,R,τ,α,μ : msg(R,Q,P,τ,μ) ∨ (task(P,τ,α) ∧ α ≠ ⊥) :: τ ⟩
    →
        gclock(T)†, gclock(T')
    || **TRUE** → gtick

Re-Verification Obligations. Obviously, the transactions satisfy the new architectural constraints (C_R5, C_R6). Since this is simply a data refinement, no new transitions are added to the program, so the program continues to satisfy the safety properties of the original specification. It is necessary to verify that the progress properties of the original program remain satisfied. In particular, we need to show that *deliver* continues to establish F18.2.1.

Proof Outline. The proof that *deliver* still satisfies F18.2.1 follows directly from the text of the program, by noting that the program maintains the following invariant, which says that all messages are located at the message's destination processor:

L1: **inv.** msg(P,Q,R,τ,μ) ⇒ P = R

which guarantees that messages sent arrive at their destination. Further, *deliver* guarantees

gclock(τ) ∧ ⟨ **set** Q,μ : msg(P,Q,P,τ,μ) :: μ ⟩ = M ∧ M ≠ { } ∧ state(P,σ)
↦
⟨ ∃ μ : ⟨ **set** Q,μ' : msg(P,Q,P,τ,μ') :: μ' ⟩ = M - { μ } :: state(P,u(P,σ,μ)) ⟩

By the definition of *msg* and L1, we have F18.2.1.

Outstanding Violations. Neither *gtick* nor *gclock* have been allocated. In addition, *task* creates the new messages at the target processor, which is a violation of C_R2.

Route messages around the ring. The write access violation by *task* can be resolved by adding a transaction type which routes messages around the ring, and then modifying *task* to create new messages locally. We use a distributed solution, allocating one routing transaction to each processor.

Refinement. We add a new transaction type *router(P)* which simply forwards messages which are not yet at their destination. Its behavior is formally described as:

C_R7: *Messages eventually reach their destination*
$msg(P,P,Q,\tau,\mu) \mapsto msg(Q,P,Q,\tau,\mu)$

C_R8: *Messages remain at their destination until they are absorbed*
$msg(P,Q,P,\tau,\mu)$ **unless** $\langle\ \forall\ R :: \neg msg(R,Q,P,\tau,\mu)\ \rangle$

The *router* transaction actually implements these new properties incrementally, by moving the messages one processor at a time, clockwise around the ring.

router(P) ≡
  Q,R,τ,μ :
    msg(P,Q,R,τ,μ), P ≠ R → msg(P,Q,R,τ,μ)†, msg(right(P),Q,R,τ,μ)
  || **TRUE** → router(P)

This transaction should be allocated to the same location as the analogous *deliver* transaction, and we must guarantee that the transaction will continue to exist whenever there is the possibility that any messages may arrive needing to be routed, e.g.,

A_R4: **inv.** $locus(deliver(P),i_1) \wedge locus(router(P),i_2) \Rightarrow i_1 = i_2$
C_R9: **inv.** $\langle\ \exists\ Q,\tau,\alpha,\mu :: (action(Q,\tau,\alpha) \wedge \alpha \neq \perp) \vee message(Q,P,\tau,\mu)\ \rangle \Rightarrow router(P)$

Re-Verification Obligations. We must show that this new transaction does not violate any of the safety properties from the original specification. In particular, we can identify the properties affected by the transaction as F5, F9, F10, and F11. The proofs are straightforward. Additionally, we must show that *router* establishes C_R7, and that this in turn allows us to conclude that F18.2.1 is satisfied by the new program.

Proof Outline. To prove that *router* establishes C_R7, we can show that the sum of the distances between the current locations of all messages in the system and their destinations never increases, and in fact will decrease. Informally, this means that eventually, all messages will arrive at their destinations. Formally, *router* establishes the following progress property:

L2: $\text{gclock}(\tau) \wedge \langle \Sigma\ P,Q,R,\tau,\mu : \text{msg}(P,Q,R,\tau,\mu) :: \text{dist}(P,R) \rangle = M \wedge M > 0$
$\mapsto$
$\text{gclock}(\tau) \wedge \langle \Sigma\ P,Q,R,\tau,\mu : \text{msg}(P,Q,R,\tau,\mu) :: \text{dist}(P,R) \rangle < M$

where *dist(P,R)* is the number of processors between *P* and *R* moving clockwise around the ring. Since by F11 messages are not created with a time stamp equal to the current time, the metric *M* cannot increase. Using L2, we can prove by induction the following progress property:

L3: $\text{gclock}(\tau) \mapsto \langle \Sigma\ P,Q,R,\tau,\mu : \text{msg}(P,Q,R,\tau,\mu) :: \text{dist}(P,R) \rangle = 0$

which states that eventually there are no messages with a current time stamp that are not at their destinations. Since by F20.1, the clock must eventually assume a value equal to the time stamp on any message, L3 gives us C_R7.

Additionally, it is clear that *deliver* establishes the following variant of F18.2.1, which states that messages which have arrived at their destination will eventually be absorbed, while allowing for the arrival of new messages (via *router*):

L4: $\text{gclock}(\tau) \wedge \langle \textbf{set}\ Q,\mu : \text{msg}(P,Q,P,\tau,\mu) :: \mu \rangle = M \wedge M \neq \{\} \wedge \text{state}(P,\sigma)$
$\mapsto$
$\text{gclock}(\tau) \wedge$
$(\langle \exists\ \mu : \langle \textbf{set}\ Q,\mu' : \text{msg}(P,Q,P,\tau,\mu') :: \mu' \rangle = M\text{-}\{\mu\} :: \text{state}(P,u(P,\sigma,\mu)) \rangle \vee$
$\text{state}(P,\sigma) \wedge M \subset \langle \textbf{set}\ Q,\mu' : \text{msg}(P,Q,P,\tau,\mu') :: \mu' \rangle)$

Since the number of messages is bounded above (by F11), eventually the set of undelivered messages with current time stamps will reach a maximum, from which time it can only decrease. C_R7 guarantees that all messages eventually reach their destinations, and L4 requires that they eventually be delivered, which gives us F18.2.1.

<u>Outstanding Violations</u>. The allocation of *gtick* and *gclock* has not yet been resolved, and the violation of C_R1 by the query for outstanding messages in *task* remains.

Refine time increment into search and update phases. As in the consensus bus example, it is clear that *gtick* and *gclock* must be distributed. However, the absence of either a global control mechanism or global memory requires that the approach be truly distributed. As a first refinement, we can break the work of *gtick* into two phases, a *search* phase which finds the earliest work, and an *update* phase which changes the current time. We find the following definitions useful at this point:

$\text{no\_msg}(R,\tau) \equiv \langle \forall\ P,Q,\mu :: \neg\text{msg}(P,Q,R,\tau,\mu) \rangle$

work(P,τ) ≡ ⟨ ∃ α,Q,R,μ :: action(P,τ,α) ∨ msg(P,Q,R,τ,μ) ⟩

Refinement. The initial program refinement is simply to split *gtick* into two global transactions which match the two phases, satisfying the following requirements:

C_R10: *The search and update phases are mutually exclusive:*
inv. ¬(gsearch ∧ ⟨ ∃ T :: gupdate(T) ⟩)

C_R11: *The search and update phases continue at least until all work is completed*
inv. work(P,T) ⇒ (gsearch ∨ ⟨ ∃ T' : T' ≤ T : gupdate(T') ⟩)

C_R12: *There is at most one time value being propagated*
inv. ⟨ Σ T : gupdate(T) :: 1 ⟩ ≤ 1

C_R13: *The search phase triggers an update phase identifying the time of the next work*
gsearch ↦ gupdate(⟨ **min** P,T : work(P,T) :: T ⟩)

C_R14: *The update phase triggers a search phase*
gupdate(T) **until** gsearch

C_R15: *The update phase changes the simulation time*
gupdate(T) **until** gclock(T)

The *gsearch* transaction is taken from property C_R13, and *gupdate* from C_R14 and C_R15.

gsearch ≡
τ : τ = ⟨ **min** P,T : work(P,T) :: T ⟩ → gupdate(τ)

gupdate(τ) ≡
T : gclock(T) → gclock(T)†, gclock(τ), gsearch

Re-Verification Obligations. We will need to prove that this pair of transactions satisfies the safety properties from the original specification. This is trivial for *gsearch* since it introduces no transitions within the original state (as a result, its characteristic property set is empty). Since *gupdate* performs the time change, it must be shown to satisfy the entire characteristic set for *gtick*, namely, F1, F7, F10, F11, F14, F15, and F21. Additionally, we must prove that the new transactions satisfy F20.1, the progress property which motivated *gtick*.

Proof Outline. The proof that the refinement continues to satisfy F20.1 involves simply proving that C_R10-C_R15 constitute a refinement of F20.1. As a review, F20.1 states:

F20.1 gclock(τ) ∧ T = ⟨ **min** P,T' : work(P,T') :: T' ⟩ ∧ T > τ
↦
⟨ ∃ T' : τ+1 ≤ T' ≤ T :: gclock(T') ⟩

Informally, C_R11 requires that if there is work to be done, then the system is in a state which, by C_R13 and C_R14, will eventually cause the clock to be moved (C_R15). More formally, C_R11 implies the following lemma, which describes the legal states of this sub-system at any given time:

L5: **inv.** $T = \langle$ **min** P,T' : work(P,T') :: T' $\rangle \Rightarrow$ (gsearch $\vee$ (gupdate(T') $\wedge$ T' < T) $\vee$ gupdate(T))

The proof follows from the general disjunction rule for **leads-to**, as follows. L5 can be used to prove L6, which says that if *T* is the next time at which there is work, then the system will eventually set the clock to time *T*, i.e.,

L6: $T = \langle$ **min** P,T : work(P,T) :: T $\rangle \mapsto$ gclock(T)

L6 follows from L5, C_R13, C_R14, C_R15, and the transitivity of **leads-to**. Since L6 implies F20.1, the refinement is proven.

Outstanding Violations. We have not yet allocated either of the new transactions or *gclock*. The read violation by *task* remains.

Distribute clock. As a first step towards allocating *gsearch*, *gupdate*, and *gclock*, we distribute the clock, giving each processor a local copy which will be updated during the *update* phase. This refinement will allow us to prove that several of the transactions satisfy the access constraints, which has not been possible until now.

Refinement. In the consensus bus example, we were able to maintain an invariant which basically required that all local clocks have the same value. Since there is no global control mechanism available in this architecture, we elect to require a slightly weaker coupling invariant. Naturally, if a solution were to present itself which enabled us to maintain all the clocks in synchrony, it would satisfy the coupling invariant. Specifically, we will define the global time to be the minimum local time value, with the added restriction that there can only be at most 2 different local time values.

C_R16: **inv.** gclock(T) $\equiv$ T = $\langle$ **min** P,T' : clock(P,T') :: T' $\rangle$

C_R17: **inv.** $1 \leq \langle \Sigma$ T : $\langle \exists$ P :: clock(P,T) $\rangle$ :: 1 $\rangle \leq 2$

The *clock* tuples are allocated to the same processor as the *task* transaction for the same node:

A_R5: **inv.** locus(clock(P,T),i_1) $\wedge$ locus(task(P,τ,α),i_2) $\Rightarrow$ $i_1 = i_2$

We re-write the transactions to use the new representation (*router* is not affected):

task(P,τ,α) ≡
  σ :
    α ≠ ⊥, clock(P,τ), ⟨ ∀ Q,R,μ :: ¬msg(R,Q,P,τ,μ) ⟩, state(P,σ)
  →
    task(P,e(P,σ,α),a(P,σ,α)),
    state(P,σ)†, state(P,s(P,σ,α)),
    ⟨ Q : c(P,Q,σ,α) :: msg(P,P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
  ‖ **NOR** → task(P,τ,α)

deliver(P) ≡
  Q,τ,μ,σ :
    clock(P,τ), state(P,σ), msg(P,Q,P,τ,μ) → msg(P,Q,P,τ,μ)†, state(P,σ)†, state(P,u(P,σ,μ))
  ‖ **TRUE** → deliver(P)

gsearch ≡
  τ : τ = ⟨ **min** P,T : work(P,T) :: T) ⟩ → gupdate(τ)

gupdate(τ) ≡
  clock(P,T) → ⟨ P :: clock(P,T)† ⟩, ⟨ P :: clock(P,τ) ⟩, gsearch

Proof Obligations. Since the refinement doesn't change the semantics of any of the transactions, it is not necessary to re-verify adherence to the original specification. This refinement allows us to prove that *deliver* only accesses local data (C_R1-C_R2), and so it satisfies the entire architectural specification.

Outstanding Violations. The allocation of *gsearch* and *gupdate* remains to be decided, and the read violation by *task* is still around.

Distribute *gupdate*. We now wish to distribute the update process to eliminate its violations of the specification. The obvious solution is to distribute the transaction, and pass control around the ring, allowing each processor to update its clock locally.

Refinement. We replace the *gupdate* transaction type with *update*. This transaction moves around the ring, updating the clock at each processor. We consider that the system is in the update phase whenever there is an *update* transaction at any site:

C_R18: gupdate(τ) ≡ ⟨ ∃ P :: update(P,τ) ⟩

Additional requirements describe the behavior of the new transaction:

C_R19: *There is at most one update transaction at a time*
$\langle \Sigma\ P,T : \text{update}(P,T) :: 1 \rangle \leq 1$

C_R20: *The transition from the search phase to the update phase takes place at processor 0*
gsearch **unless** $\langle \exists\ T :: \text{update}(0,T) \rangle$

C_R21: *The update transaction moves clockwise around the ring*
$\text{update}(P,T) \wedge P < N\text{-}1$ **until** update(right(P),T)

C_R22: *When the update transaction reaches processor N-1, the search phase begins*
update(N-1,T) **unless** gsearch

Additionally, we allocate the *update* transaction to the same processor as the local clock:
A_R6: **inv.** $\text{locus}(\text{update}(P,\tau),i_1) \wedge \text{locus}(\text{clock}(P,T),i_2) \Rightarrow i_1 = i_2$

We can now write the *update* transaction definition. The first subtransaction is simply a distributed version of the subtransaction from *gupdate*; the second subtransaction is from C_R21; and the third is from C_R22 and C_R14. This transaction now satisfies the entire specification.

update(P,τ) ≡
clock(P,T) → clock(P,T)†, clock(P,τ)
‖ P < N-1 → update(right(P),τ)
‖ P = N-1 → gsearch

We also re-write the *gsearch* transaction to satisfy C_R20 and C_R13:

gsearch ≡
$\tau : \tau = \langle$ **min** P,T : work(P,T) :: T) $\rangle \rightarrow$ update(0,τ)

Proof Obligations. We must show that the collection of *update* transactions satisfies the progress properties (C_R14 and C_R15) that were originally satisfied by *gupdate*, and that there is no violation of the characteristic safety properties for *gupdate*. The former follows directly from the transitivity of **leads-to**, and the latter is clear from examining the text of the transaction. Additionally, we can show that *update* performs only local reads and right writes, satisfying C_R1 and C_R2.

Outstanding Violations. *gsearch* has not yet been allocated. The only access violations left to resolve are those of C_R1 by *gsearch* and *task*.

Distribute search. Our solution is to distribute the search process in such a way as to guarantee that messages are delivered to their destinations before the local clock is updated. This will allow us to replace the global

message query in *task* with a local query, eliminating the access violation. We propose the creation of a *search* transaction which is passed from processor to processor. When received by a processor, the time stamp carried on the transaction should be compared against the earliest work which the processor knows about, and if the processor knows of earlier work to be done, it changes the time stamp before passing the transaction on. If the search process makes 2 passes around the ring, then we can guarantee that the delivery times for all messages are considered.

Refinement. We propose the following form for *search*:

search(P,n,T)

having the meaning: the search process is at processor P, and at this point, the earliest work found is at time T. The transaction makes 2 passes around the ring; n contains the pass number. As with the refinement to *gupdate*, we allow at most one *search* transaction to exist at any given time:

C_R23: $\langle \Sigma\ P,n,T : \text{search}(P,n,T) :: 1 \rangle \leq 1$

We introduce the obvious coupling invariant to map *gsearch* to *search*:

C_R24: **inv.** $\text{gsearch} \equiv \langle \exists\ P,n,T :: \text{search}(P,n,T) \rangle$

and we do not allow the time stamp on the *search* transaction to increase:

C_R25: $\text{search}(P,n,T)$ **unless** $\langle \exists\ P',n',T' : \text{search}(P',n',T') :: T' \leq T \rangle$

The allocation is the same one used for all entities in this process:

A_R7: $\text{locus}(\text{clock}(P,\tau),i_1) \wedge \text{locus}(\text{search}(P,n,T),i_2) \Rightarrow i_1 = i_2$

To guarantee that messages are delivered before the local clock is updated, we will want the *search* transaction to "push" messages around the ring. That is, messages should be forwarded before the transaction advances. This idea is captured formally in the following invariant, which states that on the second pass, all unreceived messages located at nodes "ahead" of the *search* transaction are in fact destined for processors "ahead" of the transaction:

C_R26: **inv.** $\text{search}(P,2,\tau) \Rightarrow (\text{msg}(Q,R,S,\tau,\mu) \wedge Q\ \text{twixt}(P,N\text{-}1) \Rightarrow S\ \text{twixt}\ (P, N\text{-}1))$

Since this invariant must hold as soon as the second pass begins, it in fact constrains the first pass, requiring that it be performed in a manner guaranteed to establish C_R26. The main significance of C_R26 is that it guarantees that the

second pass will encounter every message. We introduce the function *min_work(P)*, which simply computes the minimum time value for which work is known at processor *P*, i.e.,

$$\text{min\_work}(P) \equiv \langle \textbf{min } T' : \text{work}(P,T') :: T' \rangle$$

Additionally, we require the transaction to move clockwise around the ring, updating its time stamp to reflect the minimum time for which work has been encountered. The actual properties differ slightly depending on where the transaction currently resides:

C_R27: *During either pass, if the transaction is not at the end of the ring, it moves clockwise, changing the time if appropriate*
search(P,n,T) $\wedge$ P $\neq$ N-1 **until** search(right(P),n,min(T,min_work(P)))

C_R28: *At the beginning of the ring, pass 1 becomes pass 2*
search(N-1,1,T) **until** search(0,2,min(T,min_work(N-1)))

C_R29: *At the beginning of the ring, pass 2 becomes the update phase*
search(N-1,2,T) **until** update(0,min(T,min_work(N-1)))

Finally, we refine C_R22 to reflect the new notation:

$C_R22.1$ update(N-1,T) **unless** search(0,1,∞)

This gives us the following form for the *search* transaction:

search(P,n,T) $\equiv$
Q,R,τ,μ :
msg(P,Q,R,τ,μ), R $\neq$ P, $\tau \leq$ T $\rightarrow$ search(P,n,T)
|| **NOR**, P $\neq$ N-1 $\rightarrow$ search(right(P),n,min(T,min_work(P)))
|| **NOR**, P = N-1, n = 1 $\rightarrow$ search(0,2,min(T,min_work(P)))
|| **NOR**, P = N-1, n = 2 $\rightarrow$ update(0,min(T,min_work(P)))

The first subtransaction serves to maintain C_R26 by making the **NOR** of the other three subtransactions *false* as long as there are still messages to be forwarded (by *router*); the second, third, and fourth subtransactions are from C_R27, C_R28, and C_R29, respectively. We must also revise *update* to reflect $C_R22.1$.

update(P,τ) $\equiv$
clock(P,T) $\rightarrow$ clock(P,T)$\dagger$, clock(P,τ)
|| P < N-1 $\rightarrow$ update(right(P),τ)
|| P = N-1 $\rightarrow$ search(0,1,∞)

Finally, C_R26 implies that when the *update* transaction arrives at a node there are no messages for the transaction that are not already at the processor, that is, the following invariant is maintained by the program:

C_R30: **inv.** clock(P,τ) ∧ message(Q,R,P,τ,μ) ⇒ P = Q

This allows us to modify *task* to check for messages locally, eliminating its read violation as well:

task(P,τ,α) ≡
  σ :
    α ≠ ⊥, clock(P,τ), ⟨ ∀ Q,μ :: ¬msg(P,Q,P,τ,μ) ⟩, state(P,σ)
  →
    task(P,e(P,σ,α),a(P,σ,α)),
    state(P,σ)†, state(P,s(P,σ,α)),
    ⟨ Q : c(P,Q,σ,α) :: msg(P,P,Q,l(P,Q,τ,σ,α),v(P,Q,σ,α)) ⟩
  || **NOR** → task(P,τ,α)

Proof Obligations. We will need to show that the *search* and *update* transactions, as modified, continue to satisfy the constraints which initially motivated their form, e.g., C_R10-C_R15, and C_R19-C_R22. Additionally, we can now show that *search* does not violate the read access restrictions (C_R1), so the program now satisfies the entire architectural specification.

Proof Outline. The proof that the *update* transaction is correct is straightforward. Similarly, the invariance of C_R30 is obvious from the program text. More interesting is the proof that *search* satisfies the previous specification, in particular the progress properties C_R13-C_R15. As with the refinement of *gupdate* in the previous section, the approach is to show that C_R25-C_R29 amount to a refinement of C_R13, and that *search* satisfies the refined properties.

The proof amounts to showing that C_R26 is adequate to guarantee that the search process encounters every message before the end of the second pass. That is, if we can prove the following invariant, then the refinement is correct:

L5: **inv.** search(P,2,τ) ⇒ ⟨ ∀ Q : 0 ≤ Q < P :: τ ≤ min_work(Q) ⟩

Taking C_R26 with P = S (that is, the search process has reached the destination of some message), allows us to infer Q = S, that is, the message has reached its destination, since we have Q twixt(P,N-1) and Q twixt(R,P). From this, we can conclude (from the definition of *min_work*)

search(P,2,τ) ⇒ min_work(P) = ⟨ **min** Q,R,τ',α,μ : action(P,τ',α) ∧ α ≠ ⊥ ∨ message(R,Q,P,τ',μ) :: τ' ⟩

That is, when the search transaction arrives at node P, then the computation of *min_work* will include all messages destined for that node. Thus, each of the *min* computations in C_R27-C_R29 is guaranteed to compute the minimum for all nodes up to P, which gives us L5. C_R13 follows immediately from L5 and the transitivity of **leads-to**.

Now that all of the architectural constraints have been satisfied for all elements of the program, the process is complete. It should be noted that there is likely a one-pass algorithm for *search*, but the architectural constraints do not force us to discover it, and thus it is outside the scope of this paper.

7. Discussion

In this paper we have shown that architectural constraints can be formalized as assertions over programs and that violations of such assertions by an otherwise correct program can guide a program refinement process leading to a final solution which satisfies both the functional specifications and the architectural constraints. As in the case of specification refinement, program refinement is a creative process not likely to succumb easily to attempts at mechanization. Nevertheless, our experience demonstrates the existence of several broad classes of program refinements which, far from being mechanical in nature, entail only minimal re-verification of the program. This is a significant result since the re-verification of each new program could, in general, be a very time-consuming activity.

The first class of refinements could be termed *straight data refinement*. One example is provided by the distribution of the *gclock* tuple in the consensus bus architecture. This refinement has two characteristics which allow us to forgo most re-verification in the resulting program. First, the coupling invariant that ties the old and new data representations defines an equivalence relation and, thus, eliminates any need to re-write the specification to use the new representation. Second, the state transitions taking place in the old and new programs are also in a one-to-one relation. The resulting program thus satisfies the same safety and progress properties as the original one.

The second class of refinements are the *synchronous process distribution* refinements such as the distribution of *gtick* in the consensus bus example. No re-verification was required for this refinement because the semantics of the resulting synchronic group are identical to that of the original transaction. This refinement allowed us to convert one global transaction into a set of transactions which perform local actions but are synchronized so as to accomplish the same total effect. The resulting structure has great potential for parallel execution in a distributed ar-

chitecture which provides the needed synchronization mechanisms. The refinement as performed in the example was aided by the fact that both the original query and actions were easily partitioned among the available processors.

A third class of refinements is the *asynchronous process distribution*. The distribution of the *gdeliver* transaction (in both examples) is illustrative of this class. One global transaction was replaced by a number of asynchronous local transactions. This was possible because the processing performed by the original transaction was itself "asynchronous" in the sense that each time the original was executed, the changes it made to the global system state were in fact local and non-interfering.

The fourth class of refinements is the *serialization*, as illustrated by the final refinement in the consensus bus architecture which combines two non-interfering transactions (*task* and *deliver*) into a single transaction. This refinement is interesting because it reflects a transformation which is likely to be useful in the real world, where multiple activities may need to be grouped, either for architectural reasons, as in the example, or for other practical reasons, such as required access to shared data. The transformation described above takes advantage of the fact that the two transactions perform actions which cannot interfere with one another—*task* can only execute when there are no messages, and *deliver* has nothing to do when all messages have been delivered. Had the two transactions interfered in some way, it would have been necessary to refine one to eliminate the interference before they could be combined.

Although we made no attempt to catalog the full range of program refinements we have encountered so far in our work, it seems reasonable that a broad repertoire of useful transformations could be developed and characterized formally, along the lines of the work in the Action Systems [2] model. In this paper an attempt is made to classify program transformations by some characteristics manifest by the program to be refined. In general, their approach is to manipulate a program to reduce the interference of its statements, and then to use a "stock" refinement to distribute the computation. In effect, they move the creative part of the refinement process into the problem of removing the interference between statements; this is very similar to our approach in this paper.

Two refinements from the ring example are interesting, not so much because they can be neatly classified, but rather because they look more like specification refinements than program refinements. The refinement of *gtick*

into two phases, and the resulting refinements of the *gsearch* and *gupdate* transactions, while relatively complex, were easy to prove precisely because we were able to prove that the properties we introduced to describe the behavior of the new transactions were in fact a refinement of the original specification. This leads us to wonder if it is necessary to do program refinement at all. That is, is it possible to reason about the influence of the architectural constraints on a program without actually writing a program? This idea is attractive to us because the techniques for manipulating specifications are much better understood than those for manipulating programs, and we would like to stay within the specification realm for as long as possible. One possible approach might be to write programs simply to detect violations of the architectural constraints, but then to use the insight gained from the program to motivate the next refinement of the specification—this may turn out to be uneconomical and also to lead us back to the ad-hoc factoring of architectural constraints which characterizes specification refinements today. A more attractive alternative is to take advantage of the ability present in the Swarm logic to reason about both data (data tuples) and actions (transactions)—the specifications shown in this paper do not really draw any distinction between deriving a UNITY or a Swarm program—and to augment the specifications with assertions regarding data and action allocation to processors.

8. Conclusions

In this paper, we have shown that architectural constraints can be expressed using assertions about programs in the style of UNITY and Swarm logics—the same notation we use to write formal behavior specifications. By unifying the notations, we are able to directly factor the architectural constraints into the program derivation process. Our current approach requires a program be generated through specification refinement before the architectural constraints can be considered—the architectural constraints involve assertions over auxiliary tuples (variables) whose introduction requires the presence of a program. To the best of our knowledge, this is the first time that architectural constrains have been specified formally and compliance of the program to the architecture for which it is intended has been verified formally.

References

[1] Back, R. J. R., *Correctness Preserving Program Refinements: Proof Theory and Applications*, Mathematical Center Tracts, Amsterdam, vol. 131, 1980.

[2] Back, R. J. R., and Sere, K., "Stepwise Refinement of Parallel Algorithms," *Science of Computer Programming*, vol. 13, no. 2-3, pp. 133-180, 1990.

[3] Banâtre, J.-P., Coutant, A., and Metayer, D. L., "The Gamma Model and its Discipline of Programming," in *Science of Computer Programming*, vol. 15, pp. 55-77, 1990.

[4] Carriero, N., and Gelernter, D., "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, 1989.

[5] Chandy, K. M., and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440-452, 1979.

[6] Chandy, K. M., and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, New York, NY, 1988.

[7] Chandy, M., and Misra, J., "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 3, pp. 326-343, 1986.

[8] Cunningham, H. C., and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 365-376, 1990.

[9] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming*, Academic Press, London and New York, 1972.

[10] Dijkstra, E. D., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] Gribomont, E. P., "Stepwise Refinement and Concurrency: A Small Exercise," in *Mathematics of Program Construction*, J. L. A. v. d. Snepscheut, Eds., Springer-Verlag, New York, NY, vol. 375, pp. 219-238, 1989.

[12] Gribomont, E. P., "Development of Concurrent Systems by Incremental Transformation," in *ESOP '90, 3rd European Symposium on Programming*, N. Jones, Eds., Springer-Verlag, New York, NY, vol. 432, pp. 161-176, 1990.

[13] Gribomont, E. P., "Stepwise Refinement and Concurrency: the Finite-State Case," *Science of Computer Programming*, vol. 14, no. 2-3, pp. 185-228, 1990.

[14] Gries, D., "A note on a standard strategy for developing loop invariants and loops," *Science of Computer Programming*, vol. 2, pp. 207-214, 1982.

[15] Gries, D., and Prins, J., "A New Notion of Encapsulation," *ACM SIGPLAN NOTICES*, vol. 20, no. 6, pp. 131-139, 1985.

[16] Knapp, E., "An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 2, pp. 203-223, 1990.

[17] Lengauer, C., "A Methodology for Programming with Concurrency: the Formalism," in *Science of Computer Programming*, North-Holland, vol. 2, pp. 19-52, 1982.

[18] Morgan, C. C., "Procedures, parameters, and abstraction: separate concerns," *Science of Computer Programming*, vol. 11, pp. 17-27, 1988.

[19] Morgan, C. C., and Robinson, K., "Specification statements and refinement," *IBM Journal of Research and Development*, vol. 31, pp. 546-555, 1987.

[20] Morris, J. M., "A theoretical basis for stepwise refinement and the programming calculus," *Science of Computer Programming*, vol. 9, pp. 287-306, 1987.

[21] Morris, J. M., "Laws of data refinement," *Acta Informatica*, vol. 26, pp. 287-308, 1989.

[22] Rem, M., "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 3, pp. 251-262, 1981.

[23] Roman, G.-C., and Cunningham, H. C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1361-1373, 1990.

[24] Roman, G.-C., and Cunningham, H. C., "Reasoning about Synchronic Groups," in *Research Directions in High-Level Parallel Programming Languages*, J. P. Banâtre, D. L. Métayer, Eds., Springer-Verlag, New York, NY, vol. 574, pp. 21-38, 1992.

[25] Roman, G.-C., Gamble, R. F., and Ball, W. E., "Formal Derivation of Rule-Based Programs," Washington University, Department of Computer Science, St. Louis, Missouri. (To appear in *IEEE Transactions on Software Engineering*.), Technical Report WUCS-91-17, 1991.

[26] Wirth, N., "Program development by stepwise refinement," *CACM*, vol. 14, pp. 2321-227, 1971.

---

1 This is an example of a *constructor*, a syntactic element which occurs frequently in our notation. The general form of the constructor is:

$$\langle \textbf{op} \text{ dummy\_variables} : \text{range\_constraint} :: \text{expression} \rangle$$

where **op** is typically a binary, associative, and commutative operator (such as $+$, $*$, $\wedge$, $\vee$, written Σ, Π, $\forall$, $\exists$, respectively). Logically, the constructor creates a multi-set of values $\{v_1, v_2, \ldots, v_n\}$ by evaluating the *expression* for every possible instantiation of the *dummy_variables* satisfying the *range_constraint*. The final value of the constructor is obtained by evaluating the expression v_1 **op** v_2 **op** ... **op** v_n. If the range is empty the zero-element for the operator is returned. Other frequently used operators are *min*, *max*, and *set*, having the obvious interpretations. We occassionally omit the range of the constructor (and the first colon) when the range is obvious from the context.

2 The special predicates are **AND**, **NAND**, **OR**, **NOR**, and **TRUE**, meaning "all," "not-all,", "some," "none," and "no-matter-how-many" of the regular queries succeed.

3 The same expression, appearing in the query of a subtransaction, allows the program to query for the existence of a synchrony relation entry. Synchrony relation entries can also be deleted using the dagger. Furthermore, it is possible to query (but not modify) the transitive closure of the synchrony relation using "$\approx$" in place of "$\sim$".

4 Throughout the architectural refinements, we will use transaction names as predicates. Informally, if T is a transaction type name, then the predicate T means that there is a transaction (or collection of transactions) which satisfy the progress properties satisfied by the transaction T. This notation allows us to treat transaction refinement as a form of data refinement, a subject to which we will return in the discussion.

Index terms: stepwise refinement, formal methods, concurrency, specification, distributed simulation, architecture

Figure 1: Notation used in the Swarm proof logic.
Figure 2: One site of the postulated architecture.