Report Number: WUCS-91-05

1992-03-01

On Deriving Distributed Programs from Formal Specifications of Functional Requirements and Architectural Constraints

Gruia-Catalin Roman, C. Donald Wilcox, and Jerome Y. Plum

This design of distributed programs is a difficult task which can greatly benefit from the application of formal methods. Since design solutions are determined not only by functional requirements imposed by the application but also by the structure and behavior of the underlying hardware architecture, a complete formal treatment of the program derivation process becomes a significant challenge. The common approach is to state with a formal specification of the functional requirements and to derive the desired program through systematic refinements which factor in the architectural constraints informally, in an ad-hoc manner. This paper shows how one can employ... **Read complete abstract on page 2.**

On Deriving Distributed Programs from Formal Specifications of Functional Requirements and Architectural Constraints

Gruia-Catalin Roman, C. Donald Wilcox, and Jerome Y. Plum

Complete Abstract:

This design of distributed programs is a difficult task which can greatly benefit from the application of formal methods. Since design solutions are determined not only by functional requirements imposed by the application but also by the structure and behavior of the underlying hardware architecture, a complete formal treatment of the program derivation process becomes a significant challenge. The common approach is to state with a formal specification of the functional requirements and to derive the desired program through systematic refinements which factor in the architectural constraints informally, in an ad-hoc manner. This paper shows how one can employ a single specification method (program0wide assertions) to express both functional requirements and architectural constraints. A distributed simulation problem is used to illustrate a formal strategy for deriving a distributed program from assertions about its functionality and the constraints imposed by the choice of underlying architecture.

Washington

WASHINGTON • UNIVERSITY • IN • ST • LOUIS

School of Engineering & Applied Science

On Deriving Distributed Programs from Formal Specifications of Functional Requirements and Architectural Constraints

Gruia-Catalin Roman
C. Donald Wilcox
Jerome Y. Plun

WUCS-91-05

March 1992 (revised)

Appeared in *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992

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

Printed 3/11/93

On Deriving Distributed Programs from Formal Specifications of Functional Requirements and Architectural Constraints

Gruia-Catalin Roman, C. Donald Wilcox, and Jerome Y. Plun

Department of Computer Science
Washington University
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130

Abstract

The design of distributed programs is a difficult task which can greatly benefit from the application of formal methods. Since design solutions are determined not only by functional requirements imposed by the application but also by the structure and behavior of the underlying hardware architecture, a complete formal treatment of the program derivation process becomes a significant challenge. The common approach is to start with a formal specification of the functional requirements and to derive the desired program through systematic refinements which factor in the architectural constraints informally, in an ad-hoc manner. This paper shows how one can employ a single specification method (program-wide assertions) to express both functional requirements and architectural constraints. A distributed simulation problem is used to illustrate a formal strategy for deriving a distributed program from assertions about its functionality and the constraints imposed by the choice of underlying architecture.

1 Introduction

In his foreword to Chandy and Misra's *Parallel Program Design* [5], Hoare defines a complete theory of programming as one that includes appropriate methods for specifying programs, for reasoning about specifications, for deriving correct programs, and for transforming programs to accommodate machines available for their execution. Dijkstra provided such a theory for sequential programming [8]. Chandy and Misra's UNITY model did the same for concurrent programming. UNITY programs consist of a fixed set of variables and a fixed set of conditional multiple assignment statements. Program specification, design, and verification is supported by a proof logic given in terms of assertions of the form $\{p\}\ s\ \{q\}$ where s is universally or existentially quantified over the set of program statements while p and q are predicates over the program state. Program specifications are given in terms of assertions, and design is supported by a specification refinement methodology. Architectural constraints are dealt with by specifying a *mapping* of statements and variables to memories and processors. UNITY's treatment of mappings is informal, and therefore outside the scope of the proof logic.

This paper proposes an alternate approach for factoring the target architecture into the program design. It is based on our belief that the structural and behavioral capabilities of the underlying architecture can be formalized in terms of assertions that must be satisfied by any program executed on that particular architecture. The informal characterization that is the basis for defining program-to-architecture mappings is thus replaced by a set of proof obligations. This approach provides a new kind of formalization of the relationship between software and hardware and proposes a unified specification method for expressing both functional requirements and architectural constraints, i.e., by means of assertions. As a result, refinement methods currently in use need not differentiate between functional requirements (traditionally expressed as assertions) and architectural constraints (previously factored in informally). It now becomes possible to formally prove (rather than merely present arguments) that a particular program can execute on a particular architecture, given some software-to-hardware allocation.

The objective of this paper is to introduce the assertional approach to characterizing architectural constrains and to illustrate one way such a characterization may be used, together with assertions about program functionality, to derive a distributed program. The example employed throughout the paper is the distributed simulation of a network of sequential processors which communicate by exchanging messages, a problem which has been extensively studied and is well understood [4]. The architecture of the simulator is unconventional. It combines a synchronous multicomputer with an asynchronous one, uses partitioned memory, and provides bus-based communication.

The computational model used to represent the simulator software is a shared dataspace model of concurrency called *Swarm* [11]—a model which, unlike UNITY, allows for dynamic creation of data entities and statements and includes facilities for specifying complex mixes of synchronous and asynchronous computations. (Re-coding the algorithms stated in Swarm to conform to the language available on the target system is of no concern in this paper.) The Swarm proof logic [6, 12] (a direct extension of the UNITY proof logic) forms the basis for specifying both the functional requirements and the architectural constraints and for deriving and verifying the algorithms used in the simulation. Program derivation follows the general scheme outlined by Chandy and Misra.

The remainder of the paper consists of three main parts. Section 2 introduces the Swarm computational model and its notation. Section 3 summarizes the proof logic for Swarm and illustrates the use of assertions as a formal approach to specifying both functional requirements and architectural constraints. The former is well-understood, while the latter represents a key contribution of this paper. The functional requirements and the architectural constraints formally defined in Section 3 are used in Section 4 to derive the algorithm for the simulator.

2 Computational Model

Swarm belongs to a class of languages and models that use tuple-based communication. Other languages and models in this class are Linda [3], Associons [10], and GAMMA [2]. Two features of Swarm that are of particular importance 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, and rule-based) and architectures (e.g., synchronous, asynchronous, reconfigurable, etc.).

In Swarm the primary means for communication among the concurrent components of a 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: a *tuple space* (a finite set of data tuples), a *transaction space* (a finite set of transactions), and a *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 at any point in the computation 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 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.

In the remainder of this section, 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 $ovr(i)$. 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]

[1] This is an example of a *constructor*, a syntactic element which occurs frequently in our notation. The general form of the constructor is:
(op dummy_variables : range_constraint :: expression)
where op is typically a binary, associate, and commutative operator (such as +, *, $\wedge$, $\vee$, written Σ, Π, $\forall$, $\exists$, respectively). Logically, the constructor creates a multi-set of values $\{v_1, v_2, ..., v_n\}$ 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.

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* ($\wedge$), 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 subtransaction are evaluated in parallel, followed by the deletions and then 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:

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

[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.

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 Specification Method

Our program derivation methodology presupposes the ability to specify the operational details and formal properties of the program under development and to formalize the functional requirements and architectural constraints imposed by the application and by the available machines, respectively. The previous section reviewed the Swarm notation, which is used to describe the structure and behavior of distributed programs resulting from the application of our method. Safety and progress properties of such programs are specified and verified using the Swarm proof logic, summarized below. This same proof logic is used to specify functional requirements, by employing a Swarm-like representation of an abstraction of the system state and by using it to construct an assertional-style characterization of the class of programs which represent acceptable realizations of the particular application.

This class is further constrained by the formal specification of the architectural constraints. Their specification is not as straightforward. The difficulty rises from the fact that we are not attempting to specify the structure and behavior of a piece of hardware but the restrictions imposed on the class of programs which the particular hardware can support. As explained in Section 3.3, our architectural constraints specification has three parts: a set of restrictions on the mapping of software entities to physical loci on the architecture; a set of rules for the introduction of auxiliary variables (tuples in our logic) needed to verify compliance with the architectural constraints; and a set of assertions formulated with respect to the auxiliary variables. The assertions are proof obligations that must be met by any program executing on the particular architecture. At the same time, they supply additional assertions for use in the program derivation process.

The remainder of Section 3 presents the Swarm proof logic and applies our specification method, first to formulating properties of interest of the network being simulated (i.e., the functional requirements for the simulator); and second to formalizing architectural constraints induced by the simulator hardware.

[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 "≈" in place of "~".

3.1 Formal Foundations

For a summary of the Swarm proof logic, the reader should refer to [6, 12]. This section reviews the basic notational conventions for stating assertions about programs. These notational conventions in are summarized in Figure 1 (any property written without explicit quantification is universally quantified over all possible values of the free variables occurring in it). The notation $[t]$ denotes the predicate "transaction instance t is in the transaction space." **TRS** denotes the set of all possible transactions that could appear in some program P (not a specific transaction space), **SG** denotes the set of all synchronic groups that could appear in P, **TPS** denotes the set of all possible data tuples, and *INIT* denotes the initial state of the program P. Although we do not present any proofs, we make extensive use of the notation throughout this paper.

1. $\{p\}\ t\ \{q\}$
   The "Hoare triple" means that, whenever the precondition p is *true* and t is a transaction in the transaction space, all dataspaces that can result from executing t satisfy postcondition q.
2. p **unless** q
   If p is *true* at some point in the computation and q is not, then executing any single transaction either maintains p or establishes q.
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
   If $p \wedge \neg q$ is *true*, there exists a transaction t in the transaction space that will establish q when it is executed. The fairness assumption guarantees that t 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.

Figure 1: Notation used in the Swarm proof logic.

3.2 Expressing Functional Requirements

The example used in the paper was inspired by previous work on distributed simulation [4]. 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.

In order to start our formal specification of the requirements, we assume a global notion of time. The predicate *time(T)* is used to denote the fact that T time units have elapsed since some temporal origin. Clearly, there should always be one and only one value that makes the *time* predicate *true*, and we wish time to proceed forward one unit at a time.

inv: $\langle \Sigma T : \text{time}(T) :: 1 \rangle = 1$ (F1)

time(T) **ensures** time(T+1) (F2)

We now turn to the representation of the nodes. The predicate *action(P,τ,α)* indicates that node P is starting the execution of action α at time τ. Since an *action* predicate represents an action being started, it is true only when the current time corresponds to the execution time of the action:

inv: $\text{action}(P,\tau,\alpha) \wedge \text{time}(T) \Rightarrow \tau = T$ (F3)

Each node performs its actions on some local data. We use the predicate *data(P,τ,δ)* to indicate that a data item δ exists at node P, having been created at time τ. We make the simplifying assumption that each node uses a fixed, finite set of variables. Each variable's name is encoded, together with its value, in δ. Obviously, a datum exists only once it has been created. To reflect this in our simulation, we need to specify that the creation time of any datum is prior to the current time:

inv: $\text{data}(P,\tau,\delta) \wedge \text{time}(T) \Rightarrow \tau \leq T$ (F4)

A message with content μ sent to node P with arrival time τ is represented by the predicate *msg(P,τ,μ)*. A message exists until it has been received, i.e., the system time has reached the reception time of the message:

inv: $\text{msg}(P,\tau,\mu) \wedge \text{time}(T) \Rightarrow T < \tau$ (F5)

Upon reception, a message is removed and its content is converted to data local to the node receiving the message:

$\text{msg}(P,\tau,\mu) \wedge \text{time}(\tau\text{-}1)$ (F6)
ensures
$\neg\text{msg}(P,\tau,\mu) \wedge \text{data}(P,\tau,\mu)$

Finally, we need to specify the sequencing of actions at a node and the outcome of each action. Let *state(P)* be the state of node P. It represents all the data currently available at this node:

$\text{state}(P) = \langle$ **set** $\delta, t :$ $\text{data}(P,t,\delta) :: \delta \rangle$

Even though we represent an action only at the beginning of its execution, we consider that each *action(P,τ,α)* requires *duration(P,σ,α)*>0 units of time to execute when in state σ= *state(P)*. Executing an action at node P results in the sending of messages to other nodes, the updating of the state of node P, and the enabling of the next action to be performed at P. We introduce several functions to abstract away specific details. The function *send_msg(P,Q,σ,α)* indicates whether a message should be sent from P to Q as a result of executing action α in state σ. If a message is indeed sent, its content is computed by *msg_val(P,Q,σ,α)* and its arrival time is the execution time of the action augmented by the duration of the action and some transmission delay *msg_delay(P,Q)*. After completion of the action, the new state of node P is computed by *next_state(P,σ,α)*. The current action is always discarded upon execution, and the next action, given by *next_action(P,σ,α)*, is enabled and begins execution at time *τ+duration(P,σ,α)*. Given these definitions, the execution of an action is formally stated as a pair of progress conditions:

$\text{action}(P,\tau,\alpha) \wedge \sigma = \text{state}(P) \wedge$ (F7)
$d = \text{duration}(P,\sigma,\alpha) \wedge \sigma' = \text{next\_state}(P,\sigma,\alpha)$
ensures
$\neg\ \text{action}(P,\tau,\alpha) \wedge \sigma' = \text{state}(P) \wedge$
$\langle \forall\ Q : \text{send\_msg}(P,Q,\sigma,\alpha) ::$
$\text{msg}(Q,\tau+d+\text{msg\_delay}(P,Q),\text{msg\_val}(P,Q,\sigma,\alpha)) \rangle$

and

$\text{action}(P,\tau,\alpha) \wedge \sigma = \text{state}(P) \wedge$ (F8)
$d = \text{duration}(P,\sigma,\alpha) \wedge \alpha' = \text{next\_action}(P,\sigma,\alpha)$
$\mapsto$
$\text{action}(P,\tau+d,\alpha')$

The first condition requires that all the state changes resulting from the execution of an enabled action take place in a single step. The second condition guarantees that the next action is eventually executed.

3.3 Expressing Architectural Constraints

Now we turn to our target architecture. First, we provide an informal description of the behavior of the system; we then formalize this description using the Swarm proof logic. Our approach to the formal description of the hardware is to formulate predicates over the dataspace augmented with auxiliary variables. The latter are introduced into programs in accordance with rules concerning how the program must be augmented so as to allow one to prove that it correctly maps onto the architecture. Later, we show how some of these predicates can be used in the derivation of the program by viewing them as further constraints on the correct behavior of the system.

Architecture Overview. 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

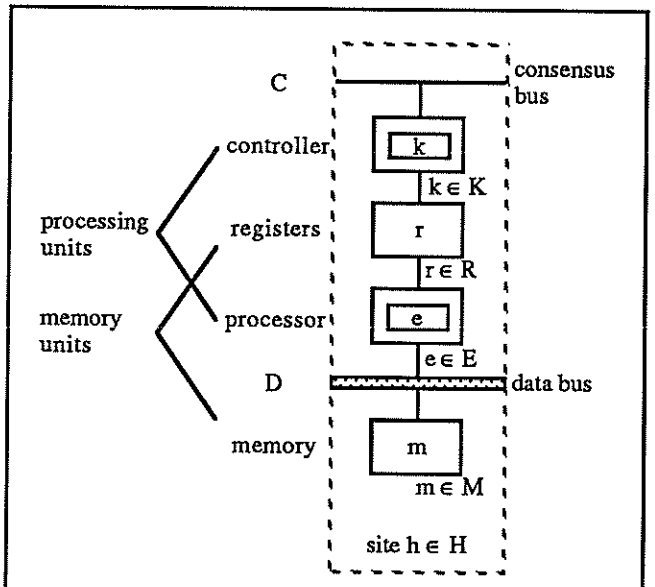

Figure 2: One site of the postulated architecture.

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. 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 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.

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*. As before, 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 introduce a function *locus* which maps program operations and data to hardware components. In Swarm, operations are represented by transactions from the transaction set **TRS** and data are represented as tuples from the set **TPS**. (**TRS** and **TPS** are implicitly defined by type definitions used in Swarm and are not discussed further in this paper.) For this reason the *locus* function is defined as:

$$\text{locus} : \mathbf{TRS} \cup \mathbf{TPS} \rightarrow K \cup R \cup E \cup M$$

and is further restricted so that transactions are mapped to processing units, and data tuples to memory units:

$$t \in \mathbf{TRS} \Leftrightarrow \text{locus}(t) \in E \cup K$$
$$v \in \mathbf{TPS} \Leftrightarrow \text{locus}(v) \in R \cup M$$

This very general definition of the locus function 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:

$$\mathbf{inv}: \quad [t_1] \wedge [t_2] \wedge t_1 \neq t_2 \Rightarrow \text{locus}(t_1) \neq \text{locus}(t_2) \qquad \text{(A1)}$$

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:

$$\mathbf{inv}: \quad [t_1] \wedge [t_2] \wedge t_1 \neq t_2 \wedge \text{locus}(t_1) \in E \Rightarrow \neg (t_1 \approx t_2) \qquad \text{(A2)}$$

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

$$\mathbf{inv}: \quad [t_1] \wedge [t_2] \wedge \text{locus}(t_1) \in K \wedge \text{locus}(t_2) \in K \Rightarrow t_1 \approx t_2 \qquad \text{(A3)}$$

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 then describe the read/write constraints in terms of invariants over these tuples. We need 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. 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 parameters for the access tuples should be computed using the *locus* function. Access constraints can be expressed in terms of three invariants.

Transactions on the processors can only read from memories located at the same site:

$$\mathbf{inv}: \quad i \in E \wedge \text{raccess}(i,j) \Rightarrow j \in R \cup M \wedge \text{site}(i) = \text{site}(j) \qquad \text{(A4)}$$

Transactions on processors can write to any memory, to registers at the same site, and to the processor itself (to change execution state):

$$\mathbf{inv}: \quad i \in E \wedge \text{waccess}(i,j) \Rightarrow \quad j \in M \vee (j \in R \cup E \wedge \text{site}(i) = \text{site}(j)) \qquad \text{(A5)}$$

Transactions on controllers can read and write only registers at the same site or the controller itself:

$$\mathbf{inv}: \quad i \in K \wedge (\text{raccess}(i,j) \vee \text{waccess}(i,j)) \Rightarrow \quad j \in K \cup R \wedge \text{site}(i) = \text{site}(j) \qquad \text{(A6)}$$

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 or a "don't care" signal 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(k)

and add the following proof obligation:

$$\mathbf{inv}: \quad k \in K \Rightarrow \neg \text{ invalid\_consensus}(k) \qquad \text{(A7)}$$

We rely here upon the earlier constraint which requires all transactions allocated to controllers to execute synchronously,

which allows us to make use of the Swarm consensus mechanism. 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 necessity, 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.

4 Program Derivation Methodology

Program derivation is the process of systematically creating a program from its specification. Work by Dijkstra [7] and Gries [9] laid a firm foundation for the use of derivation techniques in the development of sequential programs from formal problem specifications. Chandy and Misra's research on UNITY [5] has shown that a similar process is possible for concurrent programs. Their approach is to refine a formal specification until it is immediately implementable in some programming language. Alternately, Back's work with action systems [1] has centered on transforming a (typically sequential) program into a highly concurrent one. Our approach is a hybrid of these. While we start with a formal description of the problem, we move quickly to a program which is then refined, along with the specification, to achieve our final result. Our goal is not to produce, all at once, the desired program, but to identify those portions of the specification which can be implemented, and then to work on refining the remainder.

Refinements come in two flavors, data and operational. Data refinement is a process through which abstract data representations acquire an increasingly more concrete form. Operation refinement involves a gradual reduction in the granularity of the atomic operations employed by the program. Our approach requires that data and operation refinement be accompanied by a refinement of the definition for the *locus* function and the verification of the proof obligations imposed by the architectural constraints. The latter involves augmenting the operations with subtransactions which update the values of the auxiliary variables in their tuple-like representation. Since such augmentations are used only for proof purposes, they are not part of the program and are not shown whenever program segments are discussed in this section.

Another interesting detail regarding our method is a notational duality between predicates and data representation. For instance, we use *gclock(t)* to refer both to some data tuple or transaction of type *gclock* and also to a predicate interpreted to mean *"gclock(t) is in the dataspace."* Consequently, if after some refinement a version of the clock is maintained in several places (e.g., each site P contains a tuple *clock(P,t)*), *gclock(t)* can no longer be viewed as data, but can continue to be viewed as a predicate such as:

$$\text{gclock}(t) \equiv \langle \forall\ P : \text{node}(P) :: \text{clock}(P,t) \rangle$$

which can continue to be used in predicates which referred to the earlier representation of the clock. This fosters an elegant treatment of the data refinement, less reformulation of properties, and a certain degree of data-representation independence. With this introduction, we can now consider deriving the simulator program from the assertions introduced in Sections 3.2 and 3.3.

Time representation. Initially, we retain the notion of global time, and introduce a single tuple *gclock(τ)*, where τ is the global simulation time. Taking advantage of the tuple/predicate duality, we define

$$\text{time}(\tau) \equiv \text{gclock}(\tau)$$

Any transaction which manipulates the simulation time must guarantee that there is precisely one *gclock* tuple in the dataspace, i.e., (F1) should not be violated. Furthermore, some transaction must cause time to move forward in order to satisfy (F2).

Node/link state representation. The first important decision we must take is how to represent the local data residing at each node and the messages in transit along the links. One solution is to introduce two tuple types having the same parameters as the corresponding predicates of Section 3.2. However, it turns out that we need not use separate tuple types for data and messages; since the time stamp on messages is the delivery time, we can distinguish between data at P and messages destined for P by the relationship between the current global time and the time stamp on the tuple. We will represent data and messages using data tuples of type *var*, and encode the message origin in the value field. The relationship between this concrete representation and the abstract data and messages appearing in the functional requirements is given by the following definitions:

$$\text{data}(P,\tau,\beta) \equiv \text{var}(P,\tau,\beta) \wedge \text{gclock}(T) \wedge \tau \leq T$$
$$\text{msg}(P,\tau,\gamma) \equiv \text{var}(P,\tau,\gamma) \wedge \text{gclock}(T) \wedge \tau > T$$

Invariant properties (F4) and (F5) are the source of this idea; these invariants and (F6) are trivially satisfied by any program that manipulates tuples of type *var*.

Using similar reasoning and invariant (F3), we introduce the transaction type *task* to represent the sequential actions of the individual nodes along with the definition below:

$$\text{action}(P,\tau,\alpha) \equiv \text{task}(P,\tau,\alpha) \wedge \text{gclock}(\tau)$$

Initial mapping. The next step is to define the placement of data tuples and transactions on the target machine. Formally, we need to specify additional constraints on the *locus* and *site* functions which map *var* data tuples and *task* transactions onto hardware units. Data tuples are allocated to either registers or memory units and transactions to either controllers or processors, with tuples read by a particular *task* transaction being allocated to the same site as the transaction:

$$\text{locus}(\text{task}(P,\tau,\alpha)) \in E$$
$$\text{locus}(\text{var}(P,\tau,\beta)) \in M$$
$$\text{locus}(\text{task}(P, \tau_1, \alpha_1)) = \text{locus}(\text{task}(Q, \tau_2, \alpha_2)) \Leftrightarrow P = Q$$
$$\text{locus}(\text{var}(P, \tau_1, \beta_1)) = \text{locus}(\text{var}(Q, \tau_2, \beta_2)) \Leftrightarrow P = Q$$
$$\text{site}(\text{locus}(\text{task}(P,\tau_1,\alpha))) = \text{site}(\text{locus}(\text{var}(Q,\tau_2,\beta))) \Leftrightarrow P=Q$$

We also know that the clock must be allocated to some memory unit, but we postpone that decision for now.

Sequential processing simulation. To represent the interactions between an action *task(P,τ,α)* and some data value β, we postulate two predicates *ins(P,σ,β)* and *del(P,σ,β)*, and a function *val(P,σ,β)*. These predicates indicate, respectively, that a data tuple should be inserted into or deleted from the dataspace as a result of executing the action when

processor P's state is σ. In the case of an insertion, the actual content of the data tuple is computed by the function *val(P,σ,β)*. *ins*, *del*, and *val* are assumed to be an equivalent operational representation of the side effects captured by *next_state* in progress condition (F7).

We can now propose a definition for the *task* transaction, based on the progress conditions (F7) and (F8), as shown below. The introduction of Y and the insertion and deletion of *var* data tuples correspond to the computation of the new state of the processor in (F7).

```
task(P,τ,α) ≡
  σ,Y,d,α' :
    - is it time for the action?
    gclock(τ),
    - compute current state.
    σ = state(P),
    - compute duration.
    d = duration(P,σ,α),
    - compute next action.
    α' = next_action(P,σ,α),
    - collect data items.
    Y = ⟨ set T,β : var(P,T,β), T ≤ τ :: var(P,T,β) ⟩
  →
    - for each node in the neework, send messages, if required.
    ⟨ Q : send_msg(P,Q,σ,α) ::
        var(Q,τ+d+msg_delay(P,Q),msg_val(P,Q,σ,α)) ⟩,
    - delete old values.
    ⟨ T, β : var(P,T,β) ∈ Y, del(P,σ,β) :: var(P,T,β)† ⟩,
    - insert new values.
    ⟨ T, β : var(P,T,β) ∈ Y, ins(P,σ,β) :: var(P,τ,val(P,σ,β)) ⟩,
    - enable new action.
    task(P,τ+d,α')
    - recreate action if not yet enabled.
  || T : gclock(T), T ≠ τ → task(P,τ,α)
```

At this point, it is reasonable to attempt to prove that we have established properties (F7) and (F8) and to verify any proof obligations relating to the architectural constraints. We have no difficulty proving (F7) by proving a series of simple *ensures* properties which establish that all required messages are generated, the correct next state is established for the processor involved in the action, and the action is eliminated upon execution. (F8) is easily proven if we are able to show that time increases in steps of exactly one unit.

Clock distribution. Because the architecture does not provide any globally-readable memory (A4, A6), any conceivable placement of the tuple *gclock* results in a violation of the read access constraints imposed by the architecture. We distribute the global clock by placing a copy on each site *h*, either in *h.r* or *h.m*. This leads us to redefine the clock as multiple tuples *clock(P,τ)*, where *site(locus(P))* is the site where this data tuple is located. This allows us to define the global simulation time as the time value which is stored locally at each site, i.e.:

$$gclock(\tau) \equiv \langle\ \forall\ P : node(P) :: clock(P,\tau)\ \rangle$$

Since the *clock* tuple is a data tuple, it must be placed in either the registers or memories, with precisely one *clock* tuple located at each site.

$$locus(clock(P,\tau)) \in R \cup M$$
$$site(locus(clock(P,\tau_1))) = site(locus(task(P,\tau_2,\alpha)))$$

Of course, we must modify the *task* transaction to use the *clock* tuple in place of *gclock*.

Clock management. Obviously, one way of updating the simulation time is to update all clocks simultaneously. The problem is that progress condition (F8) is violated as soon as the clock advances past some action which has not yet been executed. One reasonable solution is to introduce a new transaction *gtick* which increments the clock when there are no *task* transactions to execute at the current time.

```
gtick( ) ≡
    τ: (∀ P, α : node(P) :: clock(P,τ), ¬ task(P,τ,α))
  →
    ⟨ P : node(P) :: clock(P,τ)†, clock(P,τ+1) ⟩
  || : TRUE → gtick( )
```

This transaction, of course, cannot be placed at any specific location within the hardware from which it can read all the clock tuples. This leads us to consider distributing the *gtick* transaction to each site and to attempt to use the consensus bus to achieve simultaneous updating of all local clocks. We accomplish this by introducing local *tick* transactions on the controllers, giving us the following additional constraints on the *locus* function:

$$locus(tick(P)) \in K$$
$$site(locus(tick(P))) = site(locus(task(P,\tau_2,\alpha)))$$

However, transactions located in controllers (K) can only read from registers (R). Hence it is necessary to store both the information about the clock and the non-existence of a *task* transaction in the registers. We propose to denote the existence of the transaction *task(P,τ,α)* by a data tuple *event(P,τ)* residing in the registers as indicated by the rules

$$locus(clock(P,\tau)) \in R$$
$$locus(event(P,\tau)) \in R$$
$$locus(event(P,\tau_1)) = locus(clock(P,\tau_2)))$$

and satisfying the condition

$$\mathbf{inv}:\ event(P,\tau) \Leftrightarrow \langle\ \exists\ \alpha :: task(P,\tau,\alpha)\ \rangle.$$

Initially, an *event* tuple must be created for each task.

The *event* tuples are maintained by the *task* transactions and are read by the corresponding *tick* transactions. As registers are accessible by both sets of processing units, the *task* transactions still have access to the clock and the condition for advancing the clock becomes

$$\langle\ \forall\ P, \tau : node(P) :: clock(P,\tau) \wedge \neg event(P,\tau)\ \rangle$$

By distributing this condition and by making use of the consensus feature of the synchronic group, the definition of the *tick* transactions becomes

```
tick(P) ≡
    - verify that no local action is enabled.
    τ, τ' : clock(P,τ), event(P,τ'), τ < τ' → skip
    - advance local clock when all sites have no enabled
    - actions at the current time.
  || τ : AND, clock(P,τ)† → clock(P,τ+1)
    - recreate the transaction.
  || : TRUE → tick(P)
```

with

$$\mathbf{inv}:\ tick(P) \approx tick(Q)$$

The modifications to the *task* transaction required to implement the access restrictions and to ensure the additional invariant proposed above lead to the final version of the *task* transaction shown in Figure 3 (only the lines which have been changed are annotated).

```
task(P,τ,α) ≡
  σ,Y,d,α' :
    - is it time for the action, according to the local clock?
    clock(P,τ),
    σ = state(P),
    d = duration(P,σ,α)
    α' = next_action(P,σ,α)
    Y = ⟨ set T,β : var(P,T,β), T ≤ τ :: var(P,T,β) ⟩
  →
    ⟨ Q : send_msg(P,Q,σ,α) ::
          var(Q,τ+d+msg_delay(P,Q),msg_val(P,Q,σ,α)) ⟩,
    ⟨ T, β : var(P,T,β) ∈ Y, del(P,σ,β) :: var(P,T,β)† ⟩,
    ⟨ T, β : var(P,T,β) ∈ Y, ins(P,σ,β) :: var(P,τ,val(P,σ,β)) ⟩,
    task(P,τ+d,α'),
    - notify tick(P) of the time of the next action at this site.
    event(P,τ)†, event(P,τ+d)
    - recreate action if not enabled according to local clock.
  || T : clock(P,T), T ≠ τ → task(P,τ,α)
```

Figure 3: Final version of the *task* transaction

At this point all the proof obligations are met and this brings our illustration to an end. For reasons of brevity, we avoided complete formal proofs in favor of informal arguments, although the formal proofs can be easily carried out. Moreover, this example provides an illustration of how to adopt a thorough, formal discipline in the design of complex distributed systems while avoiding (in the absence of mechanical tools) the high price associated with complete formal proofs.

5 Conclusion

In this paper we presented and illustrated a methodology for formal derivation of distributed programs. The approach uses program-wide assertions to formulate safety and progress properties of computations. Assertions are used to state functional requirements of the program as well as structural and behavioral constraints imposed by the choice of underlying architecture. It is this latter feature that contributes to the novelty of our approach. Its significance is that it offers a single, unified formal framework for integrating functional requirements and architectural constraints.

Acknowledgements. This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material. The authors thank K. C. Cox for his helpful comments on the paper.

6 References

[1] Back, R. J. R., and Kurki-Suonio, R., "Distributed Cooperation with Action Systems," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 4, pp. 513-554, 1988.

[2] 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.

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

[4] 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.

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

[6] 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.

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

[8] Dijkstra, E. W., and Fiejen, W. H. J., *A Method of Programming*, Addison-Wesley, Wokingham, England, 1988.

[9] Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1987.

[10] 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.

[11] 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.

[12] Roman, G.-C., and Cunningham, H. C., "The Synchronic Group: A Concurrent Programming Concept and its Proof Logic," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 142-149, 1990.