Reasoning about Synchronic Groups*

Gruia-Catalin Roman[1] *and H. Conrad Cunningham*[2]

[1] Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, St. Louis, MO 63130 U.S.A.
[2] Department of Computer & Information Science, University of Mississippi, 302 Weir Hall, University, MS 38677 U.S.A.

Abstract

Swarm is a computational model which extends the UNITY model in three important ways: (1) UNITY's fixed set of variables is replaced by an unbounded set of tuples which are addressed by content rather than by name; (2) UNITY's static set of statements is replaced by a dynamic set of transactions; and (3) UNITY's static $\|$-composition is augmented by dynamic coupling of transactions into *synchronic groups*. This last feature, unique to Swarm, facilitates formal specification of the mode of execution (synchronous or asynchronous) associated with portions of a concurrent program and enables computations to restructure themselves so as to accommodate the nature of the data being processed and to respond to changes in processing objectives. This paper overviews the Swarm model, introduces the synchronic group concept, and illustrates its use in the expression of dynamically structured programs. A UNITY-style programming logic is given for Swarm, the first axiomatic proof system for a *shared dataspace language*.

1 Introduction

Attempts to meet the challenges of concurrent programming have led to the emergence of a variety of models and languages. Chandy and Misra, however, argue that the fragmentation of programming approaches along the lines of architectural structure, application area, and programming language features obscures the basic unity of the programming task [4]. With the UNITY model, their goal is to unify seemingly disparate areas of programming with a simple theory consisting of a model of computation and an associated proof system.

Chandy and Misra build the UNITY computational model upon a traditional imperative foundation, a state-transition system with named variables to express the state and conditional multiple-assignment statements to express the state transitions. Above this foundation, however, UNITY follows a more radical design: all flow-of-control and communication constructs have been eliminated from the notation. A UNITY program begins

* Several of the concepts presented here appeared earlier in Roman and Cunningham's paper "The Synchronic Group: A Concurrent Programming Concept and Its Proof Logic," *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, May 1990.

execution in a valid initial state and continues infinitely; at each step an assignment is selected nondeterministically, but fairly, and executed atomically.

To accompany this simple but innovative model, Chandy and Misra have formulated an assertional programming logic which frees the program proof from the necessity of reasoning about execution sequences. Unlike most assertional proof systems, which rely on the annotation of the program text with predicates, the UNITY logic seeks to extricate the proof from the text by relying upon proofs of program-wide properties such as invariants and progress conditions.

Swarm [15] is a model which extends UNITY by permitting content-based access to data, a dynamic set of statements, and the ability to prescribe and alter the execution mode (i.e., synchronous or asynchronous) for arbitrary collections of program statements. The Swarm model is the primary vehicle for study of the *shared dataspace paradigm*, a class of languages and models in which 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. Linda [3], Associons [14], GAMMA [1], and production rule languages such as OPS5 [2] all follow the shared dataspace approach.

The Swarm design merges the philosophy of UNITY with the methods of Linda. Swarm has a UNITY-like program structure and computational model and Linda-like communication mechanisms. 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). Swarm replaces UNITY's fixed set of variables with a set of tuples and UNITY's fixed set of assignment statements with a set of transactions.

A Swarm transaction denotes an atomic transformation of the dataspace. It is a set of concurrently executed query-action pairs. A query consists of a predicate over the dataspace; an action consists of a group of deletions and insertions of dataspace elements. 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 feature adds even more dynamism and expressive power to Swarm programs. It is a relation over the set of possible transaction instances. This relation may be examined and modified by programs in the same way as the tuple and transaction spaces are. 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.

By enabling asynchronous program fragments to be coalesced dynamically into synchronous subcomputations, the synchrony relation provides an elegant mechanism for structuring concurrent computations. This unique feature facilitates a programming style in which the granularity of the computation can be changed dynamically to accommodate structural variations in the input. This feature also suggests mechanisms for the programming of a mixed-modality parallel computer, i.e., a computer which can simultaneously execute asynchronous and synchronous computations. Perhaps architectures of this type could enable both higher performance and greater flexibility in algorithm design.

This paper shows how to add this powerful capability to Swarm without compromising the ability to formally verify the resulting programs. The presentation is organized as follows. Section 2 reviews the basic Swarm notation. Section 3 introduces the notation for the synchrony relation and discusses the concept of a synchronic group. Section 4 reviews a UNITY-style assertional programming logic for Swarm without the synchrony relation and then generalizes the logic to accommodate synchronic groups. Section 5 illustrates the use of synchronic groups by means of a proof of an array summation program. Section 6 discusses some of the rationale for the design decisions.

2 Swarm Notation

The name *Swarm* evokes the image of a large, rapidly moving aggregation of small, independent agents cooperating to perform a task. This section introduces a notation for programming such computations. Beginning with an algorithm expressed in a familiar imperative notation, a parallel dialect of Dijkstra's Guarded Commands [6] language, we construct a Swarm program with similar semantics.

The program fragment given in Figure 1 (similar to the one given in [10]) sums an array of N integers. For simplicity of presentation, we assume that N is a power of 2. We also assume that elements 1 through N of the constant array A contain the values to be summed. The program introduces variables x, an N-element array of partial sums, and j, a control variable for the summing loop. The preamble of the loop initializes x to equal A and j to equal 1. Thus, the precondition Q of the program's loop is the assertion

$$pow2(N) \wedge j = 1 \wedge \langle \forall i : 1 \leq i \leq N :: x(i) = A(i) \rangle$$

where

$$pow2(k) \equiv \langle \exists p : p \geq 0 :: k = 2^p \rangle.$$

At termination, $x(N)$ is required to contain the sum of the N elements of A. Thus the postcondition R of the program is the assertion

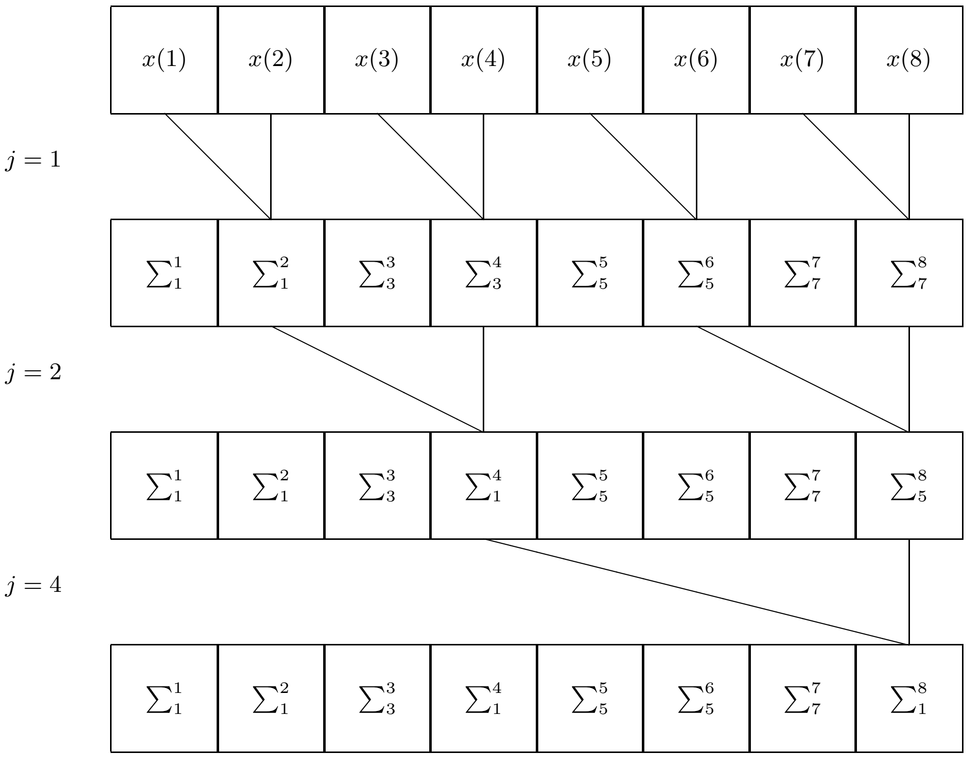
$$x(N) = sum_A(0, N)$$

where

$$sum_A(l, u) = \langle \Sigma\, k : l < k \leq u :: A(k) \rangle.$$

The loop computes the sum in a tree-like fashion as shown in the diagram: adjacent elements of the array are added in parallel, then the same is done for the resulting values, and so forth until a single value remains. The construct

$$\langle \| \;\; k : predicate :: assignment \rangle$$



$x(1)$
$x(2)$
$x(3)$
$x(4)$
$x(5)$
$x(6)$
$x(7)$
$x(8)$
$j = 1$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_3^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_7^8$
$j = 2$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_1^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_5^8$
$j = 4$
$\sum_1^1$ $\sum_1^2$ $\sum_3^3$ $\sum_1^4$ $\sum_5^5$ $\sum_5^6$ $\sum_7^7$ $\sum_1^8$

```
$j$ : **integer** ;
$x(i : 1 \le i \le N)$ : **array of integer** ;

$j := 1$ ;
$\langle\, k : 1 \le k \le N :: x(k) := A(k) \rangle$ ;
$\{\ Q\ \}$
**do** $j < N \longrightarrow\ \{\ P\ \}$
        $\langle\| \ k : 1 \le k \le N \ \wedge\ k\,\mathbf{mod}\,(j * 2) = 0 :: x(k) := x(k-j) + x(k) \rangle$ ;
        $j := j * 2$
**od**
$\{\ R\ \}$
```

Fig. 1. A Parallel Array-Summation Algorithm using Guarded Commands

is a parallel assignment command. The *assignment* is executed in parallel for each value of k which satisfies the *predicate*; the entire construct is performed as one atomic action. An invariant P for the program's loop is the assertion

$$pow2(N) \;\wedge\; pow2(j) \;\wedge\; 1 \leq j \leq N \;\wedge\; \langle \forall\, i : node(i,j) :: x(i) = sum_A(i{-}j, i) \rangle$$

where

$$node(k,l) \;\equiv\; (\, 1 \leq k \leq N \wedge k \,\mathbf{mod}\, l = 0 \,).$$

(Clearly $Q \Rightarrow P$ and $P \wedge j \geq N \Rightarrow R$.) The integer function $\frac{N}{j} - 1$ (call it vf) is an appropriate variant function to show the termination of the loop.

Swarm is a shared dataspace programming model. Instead of expressing a computation in terms of a group of named variables, Swarm uses a set of *tuples* stored in a *dataspace*. Each tuple is a pairing of a type name with a finite sequence of values; a program accesses a tuple by its content—type name and values—rather than by a specific name or address. Swarm programs execute by deleting existing tuples from and inserting new tuples into the dataspace. The *transactions* which specify these atomic dataspace transformations consist of a set of *query-action* pairs executed in parallel. Each query-action pair is similar to a production rule in a language like OPS5 [2].

How can we express the array-summation algorithm in Swarm? To represent the array x, we introduce tuples of type x in which the first component is an integer in the range 1 through N, the second a partial sum. We can express an instance of the array assignment in the **do** loop as a Swarm transaction in the following way:

$$v1, v2 : \; x(k-j, v1), x(k, v2) \quad\longrightarrow\quad x(k, v2)\dagger, x(k, v1+v2)$$

The part to the left of the $\longrightarrow$ is the query; the part to the right is the action. The identifiers $v1$ and $v2$ designate variables that are local to the query-action pair. (For now, assume that j and k are constants.)

The execution of a Swarm query is similar to the evaluation of a rule in Prolog [16]. The above query causes a search of the dataspace for two tuples of type x whose component values have the specified relationship—the comma separating the two tuple predicates is interpreted as a conjunction. If one or more solutions are found, then one of the solutions is chosen nondeterministically and the matched values are bound to the local variables $v1$ and $v2$ and the action is performed with this binding. If no solution is found, then the transaction is said to fail and none of the specified actions are taken.

The action of the above transaction consists of the deletion of one tuple and the insertion of another. The $\dagger$ operator indicates that the tuple $x(k, v2)$, where $v2$ has the value bound by the query, is to be deleted from the dataspace. The unmarked tuple form $x(k, v1 + v2)$ indicates that the corresponding tuple is to be inserted. Although the execution of a transaction is atomic, the effect of an action is as if all deletions are performed first, then all insertions.

The parallel assignment command of the algorithm can be expressed similarly in Swarm:

$$\begin{array}{l} [\, \| \;\; k : 1 \leq k \leq N \wedge k \,\mathbf{mod}\, (j * 2) = 0 :: \\ \qquad v1, v2 : x(k-j, v1), x(k, v2) \;\longrightarrow\; x(k, v2)\dagger, x(k, v1+v2) \,] \end{array}$$

Each individual query-action pair is called a *subtransaction* and the overall parallel construct a *transaction.* As with the parallel assignment, the entire transaction is executed atomically. The cumulative effect of executing a transaction is as if the subtransactions are executed synchronously: all queries are evaluated first, then the indicated tuples are deleted, and finally the indicated tuples are inserted.

In Swarm there is no concept of a process and there are no sequential programming constructs or recursive function calls. Only transactions are available. Like data tuples, transactions are represented as tuple-like entities in the dataspace. A transaction has a type name and a finite sequence of values called parameters. Transaction instances can be queried and inserted in the same way that data tuples are, but cannot be explicitly deleted. A Swarm dataspace thus has two components, the tuple space and the transaction space.

We model the execution of a Swarm program in the following way. The program begins execution with the 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 present in the transaction space at any point in time must eventually be executed (i.e., weak fairness [8]). Unless the transaction explicitly reinserts itself into the transaction space, it is deleted as a by-product of its own execution—regardless of the success or failure of its component queries. Program execution continues until there are no transactions remaining in the transaction space.

We still have two aspects of the array-summation program's **do** command to express in Swarm—the doubling of j and the conditional repetition of the loop body. Both of these actions can be incorporated into the transaction shown above. We define transactions of type *Sum* having one parameter as follows:

$$\begin{array}{lll}
Sum(j) \equiv & & \\
& [\| \; k : 1 \leq k \leq N \wedge k \,\mathbf{mod}\, (j*2) = 0 :: & \\
& \qquad v1, v2 : x(k-j, v1), x(k, v2) \;\longrightarrow\; x(k, v2)\dagger, x(k, v1+v2) \,] & \\
& \| \qquad j*2 < N \;\longrightarrow\; Sum(j*2) &
\end{array}$$

Note that the transaction above uses parameter j as a constant throughout its body. A transaction instance $Sum(j)$—representing the jth iteration of the loop—updates the set of x tuples to reflect the newly computed partial sum and inserts an appropriate transaction to continue the computation.

For a correct computation of array A's sum, the Swarm program must initialize the tuple space to contain the N elements of the array represented as x tuples, i.e., to be the set

$$\{ \; x(1, A(1)), \; x(2, A(2)), \; \cdots, \; x(N, A(N)) \; \}.$$

Similarly, the transaction space must consist of the single transaction $Sum(1)$.

Figure 2 shows a complete array-summation program. Since each x tuple is only referenced once during a computation, we modify the definition of the *Sum* subtransactions to delete both x tuples that are referenced. If a tuple form in a query is marked by the *dagger* operator, then, if the overall query succeeds, the marked tuple is deleted as a part of the action.

The first sentence in this section describes a Swarm computation with the following metaphor: "a large, rapidly moving aggregation of small, independent agents cooperating

```
**program** $ArraySum(N, A : [\exists\, p : p \geq 0 :: N = 2^p], A(i : 1 \leq i \leq N))$
**tuple types**
    $[\, i, s : 1 \leq i \leq N :: x(i, s)]$
**transaction types**
    $[\, j : 1 \leq j < N ::$
        $Sum(j) \;\equiv$
                $[\| \; k : 1 \leq k \leq N \wedge k\, \mathbf{mod}\, (j * 2) = 0 ::$
                        $v1, v2 : x(k - j, v1)\dagger, x(k, v2)\dagger \;\longrightarrow\; x(k, v1 + v2)\; ]$
                $\| \qquad j < N \qquad\qquad \longrightarrow\; Sum(j * 2)$
    $]$
**initialization**
    $Sum(1);\; [\, i : 1 \leq i \leq N :: x(i, A(i))]$
**end**
```

Fig. 2. A Parallel Array-Summation Program in Swarm

to perform a task." So far we have taken a microscopic view of Swarm computations—focusing on the small, rapidly moving, independent agents (i.e., transactions and tuples). We should not, however, ignore the macroscopic view—losing sight of the large *aggregation* of agents *cooperating* to perform a *task*. Although the *ArraySum* program does not define a process in the sense of the CSP [11] model, the evolving "swarm" of transactions does embody a distinct, purposeful activity: computing the sum of an array. Using the assertional programming logics given in Section 4, we can specify and verify such "macroscopic" properties of Swarm computations. As we see in the next section, the synchrony relation enables us to organize simple transactions into complex groups which work on portions of the overall task.

3 Synchronic Groups

The discussion in the previous section ignored the third component of a Swarm program's dataspace—the *synchrony relation*. The interaction of the synchrony relation with the execution mechanism provides a dynamic form of the $\|$ operator. The synchrony relation is a symmetric, irreflexive relation on the set of valid transaction instances. The reflexive transitive closure of the synchrony relation is thus an equivalence relation. (The synchrony relation can be pictured as an undirected graph in which the transaction instances are represented as vertices and the synchrony relationships between transaction instances as edges between the corresponding vertices. The equivalence classes of the closure relation are the connected components of this graph.) When one of the transactions in an equivalence class is chosen for execution, then all members of the class which exist in the transaction space at that point in the computation are also chosen. This group of related transactions is called a *synchronic group*. The subtransactions making up the transactions of a synchronic group are executed as if they were part of the same transaction.

The synchrony relation can be examined and modified in much the same way as the tuple and transaction spaces can. The predicate

$$Sum(i) \sim Sum(j)$$

in the query of a subtransaction examines the synchrony relation for a transaction instance $Sum(i)$ that is directly related to an instance $Sum(j)$. Neither transaction instance is required to exist in the transaction space. The operator $\approx$ can be used in a predicate to examine whether transaction instances are related by the closure of the synchrony relation.

Synchrony relationships between transaction instances can be inserted into and deleted from the relation. (The dynamic creation of a synchrony relationship between two transactions can be pictured as the insertion of an edge in the undirected graph noted above, and the deletion of a relationship as the removal of an edge.) The operation

$$Sum(i) \sim Sum(j)$$

in the action of a subtransaction creates a dynamic coupling between transaction instances $Sum(i)$ and $Sum(j)$, where i and j have bound values. If i equals j, the insertion is simply ignored. If two instances are related by the synchrony relation, then

$$(Sum(i) \sim Sum(j))\dagger$$

deletes the relationship. Note that both the synchrony relation $\sim$ and its closure $\approx$ can be tested in a query, but that only the base synchrony relation $\sim$ can be directly modified by an action. Initial synchrony relationships can be specified by putting appropriate insertion operations into the initialization section of the Swarm program.

Figure 3 shows a version of the array-summation program which uses synchronic groups. The subtransactions of $Sum(j)$ have been separated into distinct transactions $Sum(k, j)$ coupled by the synchrony relation. For each phase j, all transactions associated with that phase are structured into a single synchronic group. The computation's effect is the same as that of the earlier program.

4 Programming Logics

The Swarm computational model is similar to that of UNITY [4]; hence, a UNITY-style assertional logic seems appropriate. However, we cannot use the UNITY logic directly because of the differences between the UNITY and Swarm frameworks.

This paper follows the notational conventions for UNITY as used in [4]. Properties and inference rules are written without explicit quantification; these are universally quantified over all the values of the free variables occurring in them. This paper also introduces the notation "$[t]$" to denote the predicate "transaction instance t is in the transaction space," **TRS** to denote the set of all possible transactions (not a specific transaction space), and *Initial* to denote the initial state of the program.

First we review the proof rules for the subset of Swarm without the synchrony relation then look at how these rules can be generalized to support the synchrony relation. (For more detail on the proof rules see [5] and on the formal operational model see [15]). The Swarm programming logics have been defined so that the theorems proved for UNITY in [4] can also be proved for Swarm.

```
program $ArraySumSynch(N, A : [\exists\, p : p \geq 0 :: N = 2^p], A(i : 1 \leq i \leq N))$
tuple types
    $[\, i, s : 1 \leq i \leq N :: x(i, s)]$
transaction types
    $[\, k, j : 1 \leq k \leq N, 1 \leq j < N ::$
        $Sum(k, j) \equiv$
                        $v1, v2 : x(k - j, v1)\dagger, x(k, v2)\dagger \longrightarrow x(k, v1 + v2)$
                $\|$       $k \neq N \longrightarrow (Sum(k, j) \sim Sum(N, j))\dagger$
                $\|$       $j < N, k \,\mathbf{mod}\, (j * 4) = 0$
                             $\longrightarrow$   $Sum(k, j * 2),$
                                   $Sum(k, j * 2) \sim Sum(N, j * 2)$
    $]$
initialization
    $[\, i : 1 \leq i \leq N :: x(i, A(i))];$
    $[\, k : 1 \leq k \leq N, k \,\mathbf{mod}\, 2 = 0 :: Sum(k, 1)\, ;\, Sum(k, 1) \sim Sum(N, 1)]$
end
```

Fig. 3. A Parallel Array Summation Program Using Synchronic Groups

The "Hoare triple"

$$\{p\}\ t\ \{q\} \tag{1}$$

means that, whenever the dataspace satisfies the precondition predicate p and transaction instance t is in the transaction space, all dataspaces which can result from execution of transaction t satisfy the postcondition predicate q.

We define Swarm's **unless** relation with an inference rule similar to that given for UNITY's **unless** in [13]:

$$\frac{\langle \forall\, t : t \in \mathbf{TRS} ::\ \{p \wedge \neg q\}\ t\ \{p \vee q\}\rangle}{p\ \mathbf{unless}\ q}. \tag{2}$$

The premise of this rule means that, if p is *true* at some point in the computation and q is not, then, after the next step, p remains *true* or q becomes *true*.

Stable predicates and invariants are important for reasoning about Swarm programs. For a predicate p to be **stable** means that, if p becomes *true* at some point in a computation, it remains *true* thereafter. On the other hand, a predicate p is **invariant** if p is *true* at all points in the computation.

$$\mathbf{stable}\, p \ \equiv\ p\, \mathbf{unless}\, \mathit{false} \tag{3}$$

$$\mathbf{invariant}\, p \ \equiv\ (\mathit{Initial} \ \Rightarrow p)\ \wedge\ (\mathbf{stable}\, p). \tag{4}$$

Following UNITY's definition in [13], we define Swarm's **ensures** relation with an inference rule:

$$\frac{p\ \mathbf{unless}\ q\,, \langle \exists\, t : t \in \mathbf{TRS} ::\ (p \wedge \neg q \Rightarrow [t]) \wedge \{p \wedge \neg q\}\ t\ \{q\}\rangle}{p\ \mathbf{ensures}\ q}. \tag{5}$$

The premise of this rule means that, if p is *true* at some point in the computation, then (1) p will remain *true* as long as q is *false*, and (2) if q is *false*, there is at least one transaction in the transaction space which can, when executed, establish q as *true*. The "$p \wedge \neg q \Rightarrow [t]$" requirement generalizes the UNITY definition of **ensures** to accommodate Swarm's dynamic transaction space.

The **leads-to** relation, written

$$p \longmapsto q \tag{6}$$

means that, once p becomes *true*, q will eventually become *true*. (However, p is not guaranteed to remain *true* until q becomes *true*.) As in UNITY, the assertion $p \longmapsto q$ is *true* if and only if it can be derived by a finite number of applications of the following inference rules:

– $\dfrac{p \ \textbf{ensures}\ q}{p \ \longmapsto \ q}$ (**basis**)

– $\dfrac{p \ \longmapsto \ q, \ \ q \ \longmapsto \ r}{p \ \longmapsto \ r}$ (**transitivity**)

– *For any set W,* (**disjunction**)
$\dfrac{\langle \forall\, m \ : \ m \ \in \ W \ :: \ p(m) \ \longmapsto \ q \rangle}{\langle \exists\, m \ : \ m \ \in \ W \ :: \ p(m) \rangle \ \longmapsto \ q}$

Unlike UNITY programs, Swarm programs *terminate* when the transaction space is empty, that is

$$\mathit{Termination} \ \equiv \ \langle \forall\, t : t \in \textbf{TRS} :: \neg [t] \rangle . \tag{7}$$

How can we generalize the above logic to accommodate synchronic groups? We need to add a synchronic group rule and redefine the **unless** and **ensures** relations. The other elements of the logic are the same.

We define the "Hoare triple" for synchronic groups

$$\{p\}\ S\ \{q\} \tag{8}$$

such that, whenever the precondition p is *true* and S is a synchronic group of the dataspace, all dataspaces which can result from execution of group S satisfy postcondition q.

A key difference between this logic and the previous logic is the set over which the properties must be proved. For example, the previous logic required that, in proof of an **unless** property, an assertion be proved for all possible transactions, i.e., over the set **TRS**. On the other hand, this generalized logic requires the proof of an assertion for all possible synchronic groups of the program, denoted by **SG**.

For the synchronic group logic, we define the logical relation **unless** with the following rule:

$$\frac{\langle \forall\, S : S \in \textbf{SG} :: \ \{p \wedge \neg q\}\ S\ \{p \vee q\} \rangle}{p \ \textbf{unless}\ q} \tag{9}$$

If synchronic groups are restricted to single transactions, this definition is the same as the definition given for the earlier subset Swarm logic.

We define the **ensures** relation as follows:

$$\frac{p \ \mathbf{unless}\ q\,, \\ \langle \exists\, t : t \in \mathbf{TRS} :: (p \wedge \neg q \Rightarrow [t]) \ \wedge\ \langle \forall\, S : S \in\ \mathbf{SG}\ \wedge t \in S :: \{p \wedge \neg q\}\ S\ \{q\} \rangle \rangle}{p \ \mathbf{ensures}\ q} \quad (10)$$

This definition requires that, when $p \wedge \neg q$ is *true*, there exists a transaction t in the transaction space such that all synchronic groups which can contain t will establish q when executed from a state in which $p \wedge \neg q$ holds. Because of the fairness criterion, transaction t will eventually be chosen for execution, and hence one of the synchronic groups containing t will be executed. In the logic for the Swarm subset, the **ensures** rule requires that a single transaction be found which will establish the desired postcondition when executed. In the synchronic group logic, on the other hand, instead of requiring that a single synchronic group be found which will establish the desired state, the **ensures** rule requires that a set of synchronic groups be identified such that any of the groups will establish the desired state and that one of the groups will eventually be executed. If synchronic groups are restricted to single transactions, this definition is the same as the definition for the subset Swarm logic.

5 Example Proof

In Sections 2 and 3 we derived a Swarm program for summing an array from a similar program expressed with the Guarded Commands (GC) notation. Figure 3 gives the Swarm program *ArraySumSynch* which uses synchronic groups to compute the sum of an array. This section sketches a proof for this array sum program using the logic presented in Section 4.

The precondition for *ArraySumSynch*, call it *Initial*, is similar to Q, the precondition of the GC program's loop given in Section 2. Of course, modifications are needed to account for the differences in data and program representation. (As in the previous section, we assume that all assertions are universally quantified over all the values of the free variables occurring in them.) Using the predicates *pow*2 and *node* defined in Section 2, *Initial* can be stated as follows:

$$\begin{array}{l} pow2(N) \ \wedge\ (\, x(i,v) \ \equiv\ 1 \le i \le N \wedge v = A(i)\,) \ \wedge \\ (\, Sum(i,j) \ \equiv\ node(i, 2{*}j) \wedge j = 1\,) \ \wedge \\ (\, Sum(i,j) \sim Sum(k,l) \wedge i \le k \ \equiv\ node(i, 2{*}j) \wedge k = N \wedge j = l = 1\,) \end{array}$$

The second, third, and fourth conjuncts specify the structure of the tuple space, transaction space, and synchrony relation, respectively. Here the tuple and transaction forms, e.g., $x(i,v)$ and $Sum(i,j)$, represent predicates which are true when there is a matching entity in the dataspace and false otherwise. Likewise, predicates using the $\sim$ and $\approx$ connectives represent predicates over the synchrony relation.

The postcondition of *ArraySumSynch*, call it *Post*, is similar to R, the postcondition of the GC program. Using the sum_A expression defined in Section 2, it can be stated as

$$x(i,v) \ \equiv\ (\, i = N \wedge v = sum_A(0,N)\,).$$

To verify that the program satisfies this specification we must prove that, when the program begins execution in a state satisfying *Initial*, it eventually reaches a state satisfying *Post* and, once such a state is reached, any further execution will not falsify *Post*.

That is, we must prove a progress property and a safety property—the Sum Completion and Sum Stability properties, respectively.

Property 1 (Sum Completion) $\quad Initial \longmapsto Post$

Property 2 (Sum Stability) $\quad$ **stable** $Post$

To prove these properties, another property is needed which characterizes the unchanging relationships among the elements of the dataspace. This "structure invariant" serves a role in the Swarm proof similar to the role the loop invariant Q does in a proof of the GC program. The statement of the Structure Invariant below uses the function W_x, which is N divided by the number of x-tuples present in the dataspace, i.e.,

$$W_x = \frac{N}{\langle \# \, i, v :: x(i,v) \rangle}.$$

W_x represents the "width" of the segment of array A whose sum is in each x-tuple—a role served by the variable j in the GC program.

Property 3 (Structure Invariant)

invariant
$$\begin{array}{l} pow2(N) \ \wedge \ pow2(W_x) \ \wedge \ (\, x(i,v) \ \equiv \ node(i, W_x) \wedge v = sum_A(i - W_x, i) \,) \ \wedge \\ (\, Sum(i,j) \ \equiv \ node(i, 2{*}j) \wedge j = W_x \,) \ \wedge \\ (\, Sum(i,j) \sim Sum(k,l) \wedge i \leq k \ \equiv \ node(i, 2{*}j) \wedge k = N \wedge j = l = W_x \,) \end{array}$$

The Structure Invariant is relatively complex. This complexity arises from the mutual dependencies among the data tuples and transactions of this highly synchronous program.

¶ **Proof of the Structure Invariant.** Call this property I. To prove the invariance of I, we have to show that I holds initially and that it is stable. Since initially $W_x = N/N = 1$, $Initial \Rightarrow I$. Hence, I holds initially.

To prove I is stable we must show that I is preserved by all possible synchronic groups G, i.e., $\{ \ I \ \} \ G \ \{ \ I \ \}$ is true for arbitrary G. For any synchronic group which does not satisfy I, this predicate is trivially true. Thus, we only need to consider those synchronic groups which satisfy I. Since the value of N is not altered by any transaction, the $pow2(N)$ conjunct is preserved trivially. We now must show that each of the remaining four conjuncts is preserved.

To see that the second and third conjuncts are preserved, we note that each executing transaction deletes two x-tuples and inserts back a single x-tuple. The "indexes" (first components) of the deleted tuples are adjacent multiples of j (i.e., of W_x). The inserted tuple is positioned at the index of the rightmost deleted tuple—at a multiple of $2 * j$. The value (the second component) of the inserted tuple is equal to the sum of the values of the two deleted tuples. Furthermore, the precondition I guarantees that each of the transactions in the group operate upon different tuples.

We now consider the fourth conjunct. All transactions (allowed by I) have the same "phase" parameter j. Furthermore, the values of the "index" parameter i for these transactions are multiples of $2 * j$. Only half of the transactions, i.e., those whose index is a multiple of $4 * j$, insert successor transactions. The phase for all the inserted transactions is $2 * j$ (i.e., $2 * W_x$). As argued above, W_x is also doubled in value by the synchronic group's execution. Thus the fourth conjunct is preserved.

The only synchrony relationship for a transaction $Sum(i, j)$, for $i < N$, is with transaction $Sum(N, j)$. Upon execution, a transaction deletes this relationship. For each new transaction inserted (into phase $2 * j$), a synchrony relationship is created with $Sum(N, 2 * j)$. Thus the fifth conjunct is also preserved. ∎

¶ **Proof of Sum Stability.** We must show that the predicate *Post* is preserved by all synchronic groups allowed by the Structure Invariant I. We note that $Post \wedge I \Rightarrow W_x = N$. But $W_x = N \wedge I \Rightarrow \langle \forall\, i :: \neg node(i, 2 * W_x) \rangle$. Thus, when *Post* is true, because of the fourth conjunct of I, the transaction space must be empty. Therefore, *Post* is clearly stable. ∎

Now we can now turn our attention to the Sum Completion progress property. This large-grained progress property can be proved by induction using a finer-grained progress property corresponding to the execution of a single synchronic group. The Sum Step property is stated as an **ensures** relation.

Property 4 (Sum Step)

$$W_x = k < N \ \ \textbf{ensures} \ \ W_x = 2 * k.$$

¶ **Proof of the Sum Step property.** The proof of an **ensures** property has two parts: an existential part and an unless part.

The existential part requires us to prove that, whenever $W_x = k < N$, there is a transaction in the transaction space such that any synchronic group containing that transaction will establish $W_x = 2 * k$. But, in accordance with the Structure Invariant I, at most one synchronic group exists at a time. (Particularly, $W_x = k < N \wedge I \Rightarrow Sum(N, W_x)$.) As argued in the proof of the Structure Invariant, this synchronic group will double W_x, i.e., decrease the number of x-tuples by half.

The unless part requires us to prove $W_x = k < N$ **unless** $W_x = 2 * k$. That is, we must show for all synchronic groups G,

$$\{\ W_x = k < N \wedge I\ \}\ \ G\ \ \{\ W_x = k \vee W_x = 2 * k\ \}$$

is valid. As argued above, the only synchronic groups allowed by I will double W_x. Therefore, the **ensures** property holds. ∎

¶ **Proof of Sum Completion.** To prove Sum Completion, we must show that $\neg Post \longmapsto Post$. We note that $\neg Post \wedge I \Rightarrow W_x < N$ and $W_x = N \wedge I \Rightarrow Post$. We choose the well-founded metric $\frac{N}{W_x} - 1$. ($\frac{N}{W_x}$ is the count of the x-tuples present in the dataspace. The metric is similar to the variant function *vf* in the GC program proof.) Using this metric, the Leads-to Induction Principle [4] applied to the Sum Step property allows us to deduce that $W_x < N \longmapsto W_x = N$. Therefore, we deduce that $\neg Post \longmapsto Post$ by the Leads-to Implication Theorem [4] and the Leads-to transitivity rule. ∎

We have thus proved the Sum Stability and Sum Completion properties of the Swarm program *ArraySumSynch*. Therefore, the program satisfies its specification. Also it is true that $Post \wedge I \Rightarrow Termination$ (as we argued in the proof of Sum Stability). *ArraySum-Synch* terminates immediately upon completing the computation of the desired sum.

6 Discussion

Content-based access to data, dynamic statements (i.e., transactions), and the synchrony relation are three key features which distinguish Swarm [5, 15] from UNITY [4]. Previous papers justified the first two extensions on three grounds. First, even though the Swarm programming logic is more complex than the UNITY logic, the additional complexity is not evident in proofs unless the new features are actually used. The Swarm inference rules reduce to those employed in UNITY if we restrict the usage of Swarm to a UNITY-like subset. In such a subset, we can represent UNITY's variables as tuples, UNITY's assignments as transactions which delete and reinsert these tuples, and UNITY's static set of statements as transactions which reproduce themselves without introducing additional transactions into the dataspace. Second, for problems whose precise structure and space requirements (in terms of number of variables and statements) cannot be determined a priori, Swarm allows one to tailor the dataspace appropriately. Scaling all the values of a sparse matrix in parallel, for instance, does not require the presence of tuples and statements for the zero entries. Third, Swarm was the first notational system to make assertional-style proofs of rule-based programs feasible [9]. Moreover, since tuples can easily simulate both variables and messages, Swarm makes it possible to write and prove programs that employ these three programming paradigms.

Why introduce the synchrony relation? Consider UNITY's synchronous composition operator. The $\|$-operator binds a group of assignments into a single statement; the assignments in the group are executed synchronously as a single atomic action. Syntactically, the $\|$ is placed between the assignments making up the UNITY statement. Swarm's $\|$-operator is similar both syntactically and semantically. However, the Swarm model generalizes the notion of synchronous execution. While UNITY statements are static, anonymous entities, Swarm transactions are dynamically created entities which have unique identifiers (i.e., the type name and parameter values). This led to a more dynamic notion of synchronous execution: the atomic execution of a group of *related* transactions where the *relation* between transactions is an entity subject to examination and modification by the program. As a result, Swarm separates the definition of statements (i.e., state transformations) from the specification of their processing mode (i.e., synchronous or asynchronous). The former appears in the transaction type definitions while the latter is captured by the synchrony relation present in the dataspace. Finally, the syntactic restriction regarding the use of the $\|$-operator (i.e., no simultaneous assignments to the same variable) had to be removed to allow for synchronous execution of any group of transactions. This was accomplished by giving precedence to dataspace insertions over deletions.

Among all these changes, the ability to alter the definition of the synchrony relation is clearly the most radical departure from UNITY. Thus, the static synchrony offered by UNITY is augmented in Swarm by dynamic synchrony. This processing mode is unique to Swarm and cannot be simulated easily in UNITY. There are strong indications that dynamic synchrony will be helpful in modeling reconfigurable or heterogeneous computer architectures and will lead to new kinds of solutions to a variety of programming tasks. A related concept is also proving useful in parallel program refinement. Liu and Singh [12] have subsequently applied a complementary concept, the asynchrony relation, to the problem of refining a UNITY program toward architectures with different synchrony structures.

The capacity to model a variety of computer architectures makes Swarm attractive as a specification language for software which, because of performance considerations, is targeted to a specific and often heterogeneous hardware organization. In such cases, the software designer first derives a software specification expressed in the Swarm notation—

verifying that it is correct with respect to the application's requirements and is compatible with the chosen architecture. Subsequently, programmers use the specification as the basis for implementing the software modules allocated to the individual architectural components. To illustrate the kinds of architectures envisioned, let us consider an application in which a three-dimensional geometric model is created, manipulated, and displayed. Furthermore, let us assume that the underlying architecture consists of a producer, a transformer, and a mapper. The producer, which could be a typical workstation, runs asynchronously, generating various objects in the 3D model. The transformer is a processor pipeline that can be dynamically reconfigured and activated by the producer. When active, the transformer takes objects from the producer's memory, applies a series of transformations to them, and places the result in a very large buffer accessible to the mapper. The mapper, in turn, is an SIMD machine whose task is to copy sections of the buffer to the refresh memories of a number of display units. In Swarm, the producer activities would be specified as a set of transactions which are asynchronous with respect to each other and to the other components. The pipeline reconfiguration would involve the formation of a synchronic group consisting of transactions which manipulate, object by object, the geometric model. Finally, the SIMD machine might be captured by yet another synchronic group, one that is not subject to change. Although the sketch of the solution might be overly simplistic, this example illustrates that the synchrony relation is a convenient construct for modeling certain kinds of architectures.

Turning now to programming considerations, the first thing that must be noted is the use of the synchronic group to control the granularity of the computation. Because each synchronic group execution represents an atomic transformation of the dataspace, by adjusting the size of each group the programmer can switch between fine-grained and course-grained operations or can combine the two in a single program. Let us consider, for instance, the earlier 3D model and let us assume that it depicts a platform which holds several machines with moving parts. At each moment the position of a point in the model is affected by the combined movement of the platform, the machine to which the point belongs, and the moving part involved. Any parallel implementation of the 3D model dynamics must maintain proper consistency in the model, i.e., the effects of each movement must be perceived as a series of small atomic changes in the positions of all points on some part, on some machine, or on the entire contents of the platform. An obvious Swarm solution associates with each point three transactions, each a member of a different synchronic group. The platform group includes one transaction for each point in the model and ensures the atomicity of the coordinate transformation resulting from the platform movement. Similar synchronic groups are formed for each machine and part.

The solution is elegant because it separates the implementation of the three independent movements and requires no synchronization code. Could we have the same solution in UNITY? If the contents of the platform does not change and if moving parts do not fall off, the answer is yes. Swarm, however, can accommodate without loss of elegance the materialization and dematerialization of machines or parts, and, in general, any arbitrary structural changes in the 3D model. This is accomplished by means of appropriate restructuring of the synchronic groups. The ability to mold the computation to the changing structure of the problem at hand, through the creation and restructuring of arbitrary atomic transformations, is yet another reason for introducing synchronic groups in Swarm.

Often the capability to exchange some information among the constituent transactions of a synchronic group would make programming more convenient. Toward this end, five special predicates have been added to Swarm: **OR**, **AND**, **NOR**, **NAND**, and **TRUE**, meaning *any*, *all*, *none*, *not-all*, and *no-matter-how-many*, respectively [15]. Upon execution, each regular query (i.e., a query not containing special predicates) makes it's success/failure status available to other queries in the synchronic group. When a query which contains a special predicate is executed, this set of boolean values is accessed to determine its outcome. The result is the convenient capability to detect certain kinds of "consensus" [7] among the subtransactions of a synchronic group. Since many common programming problems involve agreement among a set of participants, this capability can be very useful. For example, quiescence within a synchronic group involving only regular queries can be detected by introducing a **NOR** query into the group. The **NOR** succeeds only when all the regular queries have failed and, therefore, no further activity can originate within the group. Of course, an **OR** query is needed to recreate the synchronic group when quiescence is not yet reached. The definition below shows the basic code structure for transactions participating in some quiescence detection activity:

```
$Worker(n) \equiv$
        $any\_work\_for\_me?$ $\longrightarrow$ $do\_the\_work$
     || **OR**                $\longrightarrow$ $Worker(n)$
     || **NOR**               $\longrightarrow$ $follow\text{-}up\_activities$
```

In contrast to the Swarm solution, classical quiescence detection algorithms are quite complex.

The updating of local clocks in a distributed simulation is another example of the use of special queries. All local clocks are placed in the same synchronic group and execute the following logic:

```
$Clock(n) \equiv$
        $is\_current\_step\_completed?$ $\longrightarrow$ **skip**
     || t: **AND**, $local\_time(n,\ t)\dagger$ $\longrightarrow$ $local\_time(n,\ t{+}1)$
     || **TRUE**                        $\longrightarrow$ $Clock(n)$
```

Components that complete their simulation early can terminate by simply removing their clock from the synchronic group. The addition and removal of one clock is hidden from all others; the only interactions among the clocks are through the special predicate "consensus" mechanism.

Although experience indicates that the synchronic group is a useful concept, a number of questions remain open. Among them, three are of immediate concern. First, does the programming convenience arising from the synchronic group feature compensate for the additional complexity of proofs of programs that employ them? Although proofs involving synchronic groups have shown, in general, to be more difficult than initially expected, drastic simplifications in program logic brought about by the use of synchronic groups may ultimately ease the verification task. Second, will the study of dynamic synchrony lead to interesting new distributed algorithms for some classical problems? Finally, how can Swarm's apparent ability to model complex architectures be put to practical benefit? On-going research will likely yield at least partial answers to these questions.

7 Conclusions

The Swarm programming logic was the first axiomatic proof system for a shared dataspace "language." Subsequently to and independently from this work, Waldinger and Stickel developed a proof theory for rule-based systems [17]. Banâtre and Le Métayer have done the same for GAMMA [1]. As far as the authors can determine, no axiomatic-style proof systems have been published for Linda.

The Swarm programming logic exploits the similarities between the Swarm and UNITY computational models. It uses the same logical relations as UNITY, but the definitions of the relations have been generalized to handle the dynamic nature of Swarm, i.e., dynamically created transactions and the synchrony relation. This paper has shown how one can extend the proof logic to accommodate the dynamic formation of synchronic groups specified by the runtime redefinition of the synchrony relation.

Acknowledgements

This work was supported by the Department of Computer Science, Washington University, Saint Louis, Missouri. The first author was also supported in part by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material. The second author thanks the Department of Computer and Information Science at the University of Mississippi for enabling him to continue this work.

References

1. J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
2. L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.
3. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
4. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.
5. H. C. Cunningham and G.-C. Roman. A UNITY-style programming logic for shared dataspace programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
7. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
8. N. Francez. *Fairness.* Springer-Verlag, New York, 1986.
9. R. F. Gamble, G.-C. Roman, and W. E. Ball. Formal verification of pure production system programs. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 339–334, July 1991.
10. W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.
11. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

12. Y. Liu and A. K. Singh. Parallel programming: Achieving portability through abstraction. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 634–640. IEEE, May 1991.
13. J. Misra. Soundness of the substitution axiom. Notes on UNITY 14–90, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, March 1990.
14. M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1981.
15. G.-C. Roman and H. C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–73, December 1990.
16. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.
17. R. J. Waldinger and M. E. Stickel. Proving properties of rule-based systems. Technical Note 494, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, December 1990.