Assertional Reasoning about Dynamic Systems

Technical Report UMCIS–1993–07

H. Conrad Cunningham
cunningham@cs.olemiss.edu

Software Methods Research Group
Department of Computer and Information Science
University of Mississippi
302 Weir Hall
University, Mississippi 38677 USA

Gruia-Catalin Roman and Jerome Y. Plun
roman@cs.wustl.edu

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, Missouri 63130-4899 USA

November 1993

A chapter to appear in the book
Parallel Computations: Paradigms and Applications,
A. Zomaya, editor, Chapman and Hall.

Assertional Reasoning about Dynamic Systems

H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi, USA

Gruia-Catalin Roman and Jerome Y. Plun
Department of Computer Science
Washington University in St. Louis, USA

Abstract

The desire to model in a straightforward manner complex features of real physical systems is often tempered by difficulties associated with reasoning about the resulting computational models. UNITY, Swarm and Dynamic Synchrony are three models of concurrency that accommodate successively more dynamic systems: from systems that can be characterized in terms of a fixed set of actions to systems involving arbitrary runtime changes in the synchronization pattern among dynamically created actions. This paper shows how, despite fundamental differences in computing styles, the three models were designed to share the same basic assertional proof logic. A sample problem, synchronous array summation, is used to illustrate the features of the three models and their impact on program verification.

1 Introduction

The need for dependability has lead to increased reliance on formal proofs of critical components and algorithms. The use of assertional-style proof techniques was formally introduced for sequential algorithms by Floyd (1967). This work was further developed by Hoare (1969) and Manna and Pnueli (1974) for proving, respectively, the partial and total correctness of the **while** construct. Assertional techniques were also used in program derivation, both for program refinement (Dijkstra 1976) and data refinement (Morris 1989).

As interest in concurrency started to grow program verification research turned its attention to concurrent programs. This issue was first tackled by Owicki and Gries (1976) who developed methods for proving partial correctness, mutual exclusion and absence of deadlock. In related but independent work, Lamport (1977) proposed the use of control predicates and introduced the terms safety and liveness. Later, Apt, Francez and de Roever (1980) extended Communicating Sequential Processes (Hoare 1978) with an axiomatic proof system, and Andrews and Reitman (1980) proposed an axiomatic proof system based on the information flow in a program. Generalized Hoare Logic was developed by Lamport (1980) to verify safety properties of concurrent programs.

Because proofs are difficult and costly, especially when they involve concurrency, techniques that simplify program verification are a welcome addition to the software designer's collection of conceptual tools. Our experience indicates that the UNITY logic

(Chandy and Misra 1988) has the potential for reducing the effort required to reason about concurrent computations. UNITY has shown that it is possible to reason about concurrent computations directly from the program text, without having to consider the possible execution sequences of the computation. A UNITY program consists of a static set of deterministic multiple-assignment statements that modify a fixed set of shared variables. The statements are executed nondeterministicly, but fairly. The UNITY proof logic is based on a subset of temporal logic (Manna and Pnueli 1992). Program properties are expressed as assertions of the form $\{p\}\ S\ \{q\}$, where S is universally or existentially quantified over the statements of the program and p and q are predicates over the data state. Safety properties characterize state transitions that may occur while progress properties characterize those that must occur.

Building upon UNITY, the Swarm model (Roman and Cunningham 1990) and its proof system (Cunningham and Roman 1990, Roman and Cunningham 1992) have extended the applicability of assertional-style proofs to content-addressable data access, to dynamic data and actions and to certain restricted forms of dynamic synchrony among actions. (Synchrony is defined as the coordinated execution of one or more actions; dynamic refers to the ability to redefine which actions are to participate in such a coordinated execution.) This allowed the use of assertional-style proofs in rule-based programming (Gamble, Roman, Ball and Cunningham in press) and made it possible to reason about programs involving multiple computing paradigms (such as shared-variable, message-passing and rule-based).

Swarm's mechanisms for specifying dynamic synchrony makes it possible to verify software that executes on reconfigurable networks consisting of a mixture of synchronous and asynchronous machines. Nevertheless, Swarm proved inadequate when faced with the more mundane task of modeling (in a simple and direct manner) the synchronization rules associated with other models of concurrency such as Concurrent Processes (Milne and Milner 1979) and Input/Output Automata (Lynch and Tuttle 1989). A rethinking of the mechanics used to specify dynamic synchrony in Swarm led to a new model, Dynamic Synchrony (Roman and Plun 1993), in which synchronization is specified by employing a synchronization predicate defined over the combined data and control state of the program. In each state, the synchronization predicate determines which actions are to be executed synchronously. The built-in synchronization rules associated with currently popular models of concurrency appear to have immediate formulations in the Dynamic Synchrony (DS) model. Furthermore, since the synchronization may change from one state to the next, highly dynamic and somewhat exotic forms of synchrony may be modeled and reasoned about in ways not previously possible.

This paper traces the impact dynamic features have on the complexity of the assertional-style proof logics for the three models (UNITY, Swarm and DS) discussed above. UNITY (section 2) has a static structure composed of a fixed set of actions (called statements) with some of the actions consisting of synchronously executed subactions. In Swarm (section 3), actions (called transactions) are created dynamically but can be deleted only as a side effect of their execution and synchronously executed (disjoint) groups of actions may be constructed at runtime. DS (section 4) allows arbitrary deletion of actions and arbitrary formation of synchronously executed groups of actions. Variants of a single example—parallel synchronous array summation—are used to demonstrate differences in verification style associated with the three models. The programming solution is essen-

tially the same in all three cases. While this choice does not emphasize the differences in expressive power among the three models, it does make it easier for the reader to shift from one notation to the next and to gain insight in the way the proof obligations change from one model to the next. Conclusions and future work are discussed in section 5.

2 Static statements

2.1 Overview of UNITY

Attempts to meet the challenges of concurrent programming have led to the emergence of a variety of models and languages. Chandy and Misra (1988), 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. 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. Above this foundation, however, UNITY follows a more radical design: all flow-of-control and communication constructs have been eliminated from the notation. For the purposes of this paper, a UNITY program consists of three sections:

1. the declarations for a finite, static set of variables,
2. the descriptions of the allowed initial values for the variables and
3. a finite, static set of statements.

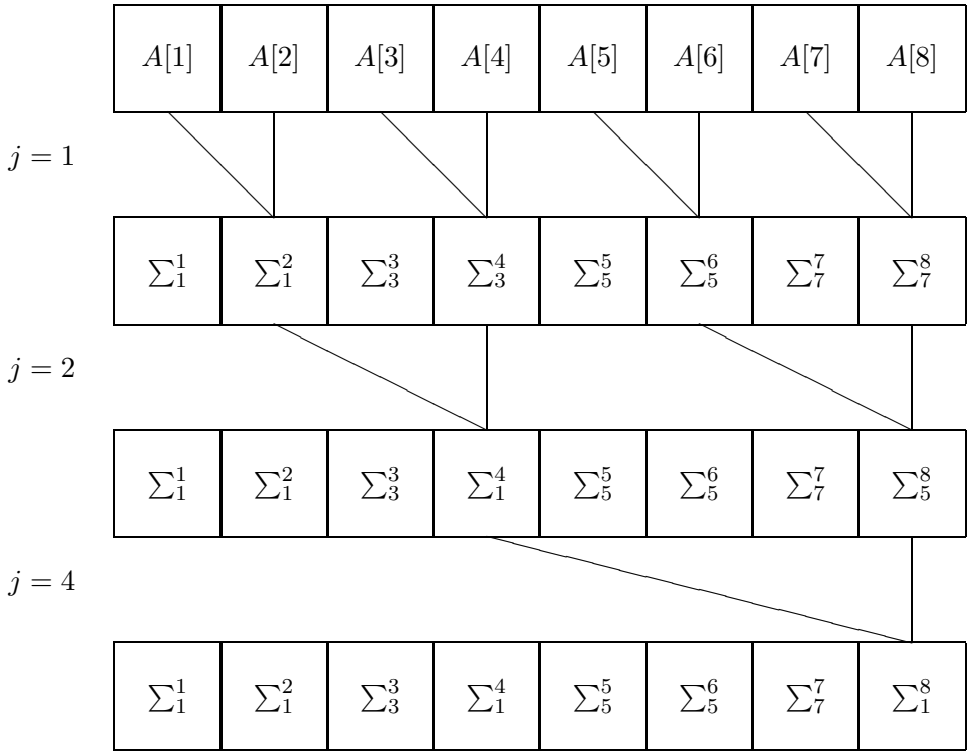
The only type of statement allowed is the conditional, multiple-assignment statement; each statement specifies a deterministic, terminating, atomic change to the program's state (i.e. to the values of the variables).

The execution of a UNITY program starts in any state that satisfies the initial condition and continues for an infinite number of steps. In each step the execution mechanism nondeterministically selects one of the assignment statements and executes it. The selection must be fair in the sense that every statement is executed infinitely many times during an infinite execution—weak fairness as defined by Francez (1986).

To illustrate each of the programming models, this paper uses the problem of summing an array $A[1..N]$ of integers. For simplicity, we assume that N is a power of 2. Thus a precondition for a UNITY program is the assertion $pow2(N)$ where

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

Note: We write quantified expressions in the form $\langle Op\ x : R(x) :: T(x) \rangle$ where Op is the quantification operator, x is a list of dummy variables whose scopes are delimited by the angle brackets, $R(x)$ is a predicate giving the range of values for the dummies over which the quantification is to be done and $T(x)$ is the term to which the operation is to be applied. Op is a commutative and associative operator such as $\exists$ ("there exists"), $\forall$ ("for all"), Σ ("sum of") or $\#$ ("number of").



program $U\_Sum(N, A)$
  N : **integer constant** { $\langle \exists\, p : p \geq 0 :: N = 2^p \rangle$ },
  A : **array** $[1..N]$ **of integer constant**
declare
  x : **array** $[1..N]$ **of integer**,
  j : **integer**
initially
  $j, x = 1, A$
assign
  $\langle \| \; k : 1 \leq k \leq N \wedge k \; \mathbf{mod} \; (2 * j) = 0 ::$
    $x[k] := x[k-j] + x[k] \rangle$
  $\|$ $\quad j := 2 * j$ **if** $j < N$
end

Figure 1: Array summation in UNITY

Eventually, after all N elements of A have been added, we want some program variable to contain their sum. That is, we want the assertion

$$additions\_done \ \wedge \ result = sum_A(0, N)$$

to hold for some appropriate predicate *additions_done* and variable *result*, where

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

One parallel algorithm for this problem is to compute the sum in a tree-like fashion as shown in the diagram in Fig. 1: 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. To develop a program for this multi-phase algorithm, we introduce program variables x, an N-element array to hold the partial sums that are computed, and j, an integer giving the width of the segment of array A in each partial sum. Before and after each phase we require that the assertion

$$pow2(N) \ \wedge \ pow2(j) \ \wedge \ j \leq N \ \wedge \ \langle \forall \, i : node(i, j) :: x[i] = sum_A(i-j, i) \rangle$$

hold, where

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

To establish this assertion at the beginning of the algorithm, we initialize x to equal A and j to equal 1. To achieve the tree-like computation, the program should double the value of j for each successive phase. Note that when $j = N$, all needed additions have been completed and $x[N]$ contains the final result. Thus we let $j = N$ replace the *additions_done* predicate and $x[N]$ replace the *result* variable. Figure 1 shows the resulting UNITY program for summing array A.

2.2 Proof logic

To accompany the simple but innovative UNITY model, Chandy and Misra (1988) have formulated an assertional programming logic that 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. The UNITY programming logic uses the logical relations **unless**, **stable**, **invariant**, **constant** and $\longmapsto$ (read "leads-to"). It defines these relations in terms of the set of program statements and the Hoare triple (Hoare 1969). In this section, **Prog** denotes the set of assignment statements of the program under consideration and *Initial* denotes the initial condition predicate. As in Chandy and Misra (1988), properties and inference rules are written without explicit quantification; these are universally quantified over all the values of the free variables occurring in them.

In terms of Dijkstra's weakest precondition calculus (Dijkstra 1976), the UNITY logic defines the Hoare triple $\{p\} \ S \ \{q\}$ for some statement S in **Prog** such that

$$\{p\} \ S \ \{q\} \ \equiv \ (\, p \Rightarrow wp(S, q) \,).$$

Informally, the triple means that, whenever statement S is executed in a state satisfying precondition predicate p, the next state will satisfy the postcondition predicate q.

Misra (1990) defines UNITY's **unless** relation with the inference rule:

$$\frac{\langle \forall\, S : S \in \mathbf{Prog} :: \{p \wedge \neg q\}\ S\ \{p \vee q\}\rangle}{p\ \mathbf{unless}\ q}$$

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, invariant and constant properties are important for reasoning about UNITY programs. For a predicate p to be **stable** means that, if p becomes *true* at some point in a computation, it remains *true* thereafter. A predicate p is **invariant** if p is *true* at all points in the computation. A predicate p is **constant** if it either always remains *true* or always remains *false*.

$$\begin{array}{ll} \mathbf{stable}\ p & \equiv p\ \mathbf{unless}\ \mathit{false} \\ \mathbf{invariant}\ p & \equiv (\mathit{Initial}\ \Rightarrow p)\ \wedge\ (\mathbf{stable}\ p) \\ \mathbf{constant}\ p & \equiv (\mathbf{stable}\ p)\ \wedge\ (\mathbf{stable}\ \neg p) \end{array}$$

A useful, although somewhat controversial (Misra 1990, Sanders 1991), rule is the Invariant Substitution Axiom. This axiom states that if $x = y$ is an invariant of **Prog**, then x can replace y in all properties of **Prog**. In particular, if I is an invariant, then $I = \mathit{true}$ is also an invariant and, hence, I can replace *true* in any property and vice versa.

Misra (1990) also defines UNITY's **ensures** relation with inference rule:

$$\frac{p\ \mathbf{unless}\ q\,,\ \langle \exists\, S : S \in \mathbf{Prog} :: \{p \wedge \neg q\}\ S\ \{q\}\rangle}{p\ \mathbf{ensures}\ q}$$

The premise of this rule means that, if p is *true* at some point, then (1) p will remain *true* as long as q is *false*, and (2) if q is *false*, there is at least one statement whose execution will always make q *true*. Because of fairness, that statement must eventually be executed.

The **leads-to** relation, written

$$p \longmapsto q$$

means that, once p becomes *true*, q will eventually become *true*. (However, p is not guaranteed to remain *true* until q becomes *true*.) 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:

- $$\frac{p\ \mathbf{ensures}\ q}{p \longmapsto q}$$ (**basis**)

- $$\frac{p \longmapsto q,\quad q \longmapsto r}{p \longmapsto r}$$ (**transitivity**)

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

UNITY programs do not terminate in the traditional sense. However, UNITY programs may reach a fixed point, a state that is repeated infinitely whenever it is reached during an execution. The fixed point predicate *FP* characterizes these states. We can construct *FP* syntactically as the conjunction of the equations corresponding to the assignment statements of the program. That is, a program with the **assign** section

$$X := E \textbf{ if } B \ [] \ Y := F \textbf{ if } C$$

has the *FP* predicate

$$(B \Rightarrow X = E) \wedge (C \Rightarrow Y = F).$$

2.3 Example proof

To prove the UNITY array summation program given in Fig. 1, we must first state its specification formally. The initial condition for this program, the predicate *Initial*, can be stated as

$$pow2(N) \ \wedge \ j = 1 \ \wedge \ x = A.$$

The desired "postcondition" for the program, the predicate *Post*, can be stated as

$$j = N \ \wedge \ x[N] = sum_A(0, N).$$

Whenever the program begins execution in a state satisfying *Initial*, it must eventually reach a state satisfying *Post* and, once such a state is reached, any further execution must 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) $\mathit{Initial} \longmapsto \mathit{Post}$

Property 2 (Sum Stability) **stable** *Post*

Proof of Sum Stability: Note that if $j = N$ then none of the assignments change the state. Thus *Post* is preserved by all statements of the program. ∎

To prove the Sum Completion property, we must first characterize the relationships among the values of the program variables N, j, A and x. In the informal derivation of the program given in section 2.1, we identified an assertion that must hold "before and after each phase" of the computation. Since each phase of the computation corresponds to the execution of one atomic statement in the program, the conjuncts of the assertion correspond to the UNITY invariants we need. We call these the Phase Invariants.

Property 3 (Phase Invariants)

invariant $pow2(N)$
invariant $pow2(j)$
invariant $j \leq N$
invariant $\langle \forall\, i : node(i, j) :: x[i] = sum_A(i-j, i) \rangle$

Proof of the Phase Invariants: Let I_1, I_2, I_3 and I_4 denote the first, second, third and fourth Phase Invariants, respectively.

To prove that a predicate is invariant we must show that the predicate holds initially and that it is stable (i.e. each statement preserves the predicate). Given the precondition assumption about N and the initialization of j to 1 and x to A, it is easy to see that each of the four predicates hold initially.

The proofs of stability are straightforward. I_1 follows from the declaration of N as a constant. I_2 follows from the observation that doubling is the only change that can be made to j. To prove I_3 we apply the Invariant Substitution Axiom with invariants I_1 and I_2. Thus, noting the guard on the "$j := 2 * j$" assignment, I_3 follows from the validity of

$$I_1 \wedge I_2 \wedge I_3 \wedge j < N \;\Rightarrow\; 2 * j \leq N.$$

Similarly, using the first three invariants and noting the range predicate on the assignments to x, invariant I_4 follows from the validity of

$$I_1 \wedge I_2 \wedge I_3 \wedge I_4 \;\Rightarrow\; \langle \forall\, i : node(i, 2 * j) :: x[i-j] + x[i] = sum_A(i-2 * j, i) \rangle.$$ ∎

Now we can turn our attention to the Sum Completion progress property. This is a large-grained progress property for which leads-to induction (Chandy and Misra 1988) seems to be a promising proof technique. To use this technique, we must find a finer-grained leads-to property that can be composed with itself transitively to deduce the large-grained leads-to. We must also find a metric function that maps the program state to a well-founded set; the value of this metric must decrease for each successive transitive application of the finer-grained leads-to property.

Proof of Sum Completion: To prove Sum Completion, we must show that $\neg Post \longmapsto Post$. We proceed by leads-to induction choosing the integer function

$$\frac{N}{j}$$

as the metric. This gives us two proof obligations:

- **invariant** $\neg Post \;\Rightarrow\; 1 \leq j \leq N$ to guarantee that the metric is well-founded,
- $\neg Post \wedge j = k \;\longmapsto\; (\neg Post \wedge j > k) \vee Post$ to guarantee the progress.

Using the Invariant Substitution Rule and the Phase Invariants, we see that the first of these obligations holds and that the second obligation follows from

$$j = k < N \;\longmapsto\; j > k.$$

Using the basis rule of the $\longmapsto$ definition, this property follows from the Sum Step property stated and proved below. ∎

The Sum Step property is a fine-grained progress property corresponding to the execution of a single statement. We state it as an **ensures** relation.

Property 4 (Sum Step) $j = k < N$ **ensures** $j = 2 * k$

Proof of the Sum Step property: The proof of an **ensures** property has two parts: an unless part and an existential part. Since the only program variable referenced in this property is j, we only need to consider the assignments to j.

Because of the definition of **unless**, $j = k < N$ **unless** $j = 2 * k$ follows from the validity of

$$j = k < N \wedge j \neq 2 * k \wedge j < N \;\Rightarrow\; 2 * j = k < N \vee 2 * j = 2 * k.$$

The existential part requires us to prove there is a statement that will always establish $j = 2 * k$ when $j = k < N \wedge j \neq 2 * k$ holds before execution. Clearly, the doubling operation on j is the only assignment that can change the value of j. Thus the existential part follows from the validity of

$$(j = k < N \wedge j \neq 2 * k \wedge j < N \;\Rightarrow\; 2 * j = 2 * k) \wedge$$
$$(j = k < N \wedge j \neq 2 * k \wedge j \geq N \;\Rightarrow\; 2 * j = 2 * k).$$

Note that the second implication above is true because its left-hand-side is false. ∎

We have thus proved the Sum Stability and Sum Completion properties of the UNITY program. Therefore, the program satisfies its specification. Furthermore, by inspection of the UNITY program, we see that its fixed point predicate *FP* is

$$j = 0 \;\vee\; j \geq N.$$

If we let *PI* denote the conjunction of the Phase Invariants, a bit of logical and arithmetic manipulation will show that

$$FP \wedge PI \;\equiv\; Post \wedge PI.$$

Hence, those states in which the program has reached a fixed point are exactly those states in which the predicate *Post* is true.

3 Dynamic transactions

3.1 Overview of Swarm

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 dynamic computations. Beginning with the UNITY program given in Fig. 1, we construct a program in the Swarm notation with similar semantics.

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 that 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 (Brownston, Farrell, Kant and Martin 1985).

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 UNITY program 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 (Sterling and Shapiro 1986). 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 to array x in the UNITY program can be expressed similarly in Swarm:

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

Each individual query-action pair is called a subtransaction and the overall parallel construct a transaction. As with the UNITY 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 a way similar to UNITY. 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. Unless the transaction explicitly reinserts itself into the transaction space, it is deleted as a by-product of its

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

Figure 2: Swarm encoding of the UNITY array summation program

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 UNITY program to express in Swarm—the doubling of j and the repetitive execution of statements. Both of these actions can be incorporated into the transaction shown above. We define transactions of type *Sum* having one parameter as follows:

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

Note that the transaction above uses parameter j as a constant throughout its body. A transaction instance $Sum(j)$—representing the jth phase of the algorithm—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 † operator, then, if the overall query succeeds, the marked tuple is deleted as a part of the action.

So far, we have ignored a 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 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 that 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 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 that 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.

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

Figure 3: Array summation in Swarm using synchronic groups

3.2 Proof logic

Even though, at an abstract level, the Swarm computational model is similar to that given for UNITY in section 2, the Swarm proof logic must be able to accommodate the dynamic nature of the Swarm dataspace. First we consider 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. More detail on the operational model (Roman and Cunningham 1990) and proof rules (Cunningham and Roman 1990, Roman and Cunningham 1992) are given elsewhere. The Swarm programming logics have been defined so that the theorems proved for UNITY in Chandy and Misra (1988) can also be proved for Swarm.

For any transaction t in the program, the Swarm logic defines the Hoare triple

$$\{p\}\ t\ \{q\}$$

such that, whenever the dataspace satisfies the precondition predicate p and transaction instance t is in the transaction space, all dataspaces that 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 section 2:

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

In this section **TRS** denotes the set of all possible transactions (not a specific transaction space). As in the UNITY logic, 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*.

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

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

Here we use the notation "$[t]$" to denote the predicate "transaction instance t is in the transaction space". 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 that, when executed, will always 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.

For the Swarm logic, we define the **stable**, **invariant**, **constant** and $\longmapsto$ as we did in section 2 for UNITY. However, unlike UNITY programs, Swarm programs terminate when the transaction space is empty, that is

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

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.

For any synchronic group S in the program, the Swarm logic defines the Hoare triple

$$\{p\}\ S\ \{q\}$$

such that, whenever precondition p is *true* and S is a synchronic group of the dataspace, all dataspaces that 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:

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

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:

$$\frac{\begin{array}{c} p \textbf{ unless } q\,, \\ \langle \exists\, t : t \in \textbf{TRS} :: (p \wedge \neg q \Rightarrow [t]) \ \wedge\ \langle \forall\, S : S \in \ \textbf{SG} \ \wedge t \in S :: \{p \wedge \neg q\}\ S\ \{q\}\rangle\rangle \end{array}}{p \textbf{ ensures } q}$$

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 that 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 that 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 that 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.

3.3 Example proof

In section 3.1 we derived a Swarm program for summing an array from a similar program expressed in the UNITY notation. Figure 3 gives a Swarm program that uses synchronic groups to compute the sum of an array. This section sketches a proof for this array summation program.

The initial condition predicate for the program in Fig. 3 is similar to the initial condition of the UNITY program given in Fig. 1. Of course, modifications are needed to account for the differences in data and program representation. (As in the previous sections, 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, the Swarm program's predicate *Initial* can be stated as follows:

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

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 that 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. Note how the $\equiv$ connective is used in conjunction with the universal quantification of the free variables to specify the exact structure of the dataspace.

The postcondition of the Swarm program is similar to the "postcondition" of the UNITY program. Using the sum_A expression defined in section 2, the Swarm program's predicate *Post* can be stated as

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

In terms of the general requirements stated in section 2.1, we use the tuple $x(N,v)$ to replace the variable *result* and the test for the presence of exactly one x-tuple to replace the *additions_done* predicate.

As with the UNITY program, to verify that the Swarm 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 must not falsify *Post*. That is, we must prove the Sum Completion and Sum Stability properties as we did for the UNITY program.

Property 5 (Sum Completion) $\mathit{Initial} \longmapsto \mathit{Post}$

Property 6 (Sum Stability) **stable** *Post*

To prove these properties, we must ultimately reduce them to simple **unless** and **ensures** properties that must themselves be proved over all elements of the set **SG**. In general, proofs over this set are quite difficult. Fortunately, the Invariant Substitution Axiom can make these proofs easier. First we find and prove invariants that characterize the unchanging structure of the transaction space and synchrony relation. Then we use these invariants to simplify the proofs of other properties—we can ignore synchronic group and transaction space configurations that do not satisfy the invariants.

Because this is a highly synchronous program, the elements of the tuple space, transaction space and synchrony relation are mutually dependent. Because of this mutual dependency, we state the relationships among them with a single invariant called the Structure Invariant. This invariant characterizes the unchanging relationships among the elements of the dataspace in a manner similar to the way the four Phase Invariants did for variables of the UNITY 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 UNITY program.

Property 7 (Structure Invariant)

invariant
$\mathit{pow2}(N) \ \wedge \ \mathit{pow2}(W_x) \ \wedge \ (\, x(i,v) \ \equiv \ \mathit{node}(i, W_x) \wedge v = \mathit{sum}_A(i - W_x, i) \,) \ \wedge$
$(\, \mathit{Sum}(i,j) \ \equiv \ \mathit{node}(i, 2{*}j) \wedge j = W_x \,) \ \wedge$
$(\, \mathit{Sum}(i,j) \sim \mathit{Sum}(k,l) \wedge i \leq k \ \equiv \ \mathit{node}(i, 2{*}j) \wedge k = N \wedge j = l = W_x \,)$

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 $W_x = N/N = 1$, $\mathit{Initial} \Rightarrow I$. 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 that does not satisfy I, this predicate is trivially true. Thus, we only need to consider those synchronic groups that satisfy I. Since the value of N is not altered by any transaction, the $\mathit{pow2}(N)$ 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 turn our attention to the Sum Completion progress property. In a manner similar to the UNITY program, we prove this large-grained progress property by induction using a finer-grained progress property corresponding to the execution of a single synchronic group.

Proof of Sum Completion: To prove Sum Completion, we must show that $\neg Post \longmapsto Post$. We proceed by leads-to induction.

We note that $\neg Post \wedge I \Rightarrow 1 < W_x < N$, that $W_x = N \wedge I \Rightarrow Post$, and that W_x increases upon the execution of a synchronic group. Thus we choose the well-founded metric $\frac{N}{W_x}$. ($\frac{N}{W_x}$ is the count of the x-tuples present in the dataspace. The metric is similar to the one used in the UNITY program's Sum Completion proof.)

Thus the remaining proof obligation is

$$\neg Post \ \wedge \ W_x = k \ \longmapsto \ (\neg Post \wedge W_x > k) \ \vee \ Post.$$

Using the Invariant Substitution Axiom and the Structure Invariant, we see that this follows from

$$W_x = k < N \ \longmapsto \ W_x > k.$$

Using the basis rule of the $\longmapsto$ definition, this property, in turn, follows from the Sum Step property stated and proved below. ∎

The Sum Step property is a fine-grained property corresponding to the execution of a single synchronic group. We state it as an **ensures** relation.

Property 8 (Sum Step) $W_x = k < N$ **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,

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

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

We have thus proved the Sum Stability and Sum Completion properties of the Swarm program. 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). Thus the Swarm program terminates immediately upon completing the computation of the desired sum.

Differences between the proofs for the UNITY and Swarm programs stem from three principal sources. First, the dynamic nature of the Swarm data tuples leads to the inclusion of invariant properties taken for granted in UNITY where variables cannot disappear or acquire multiple values. Second, the Swarm program allows a termination proof. Third, the dynamic nature of synchronic groups forced us to strengthen the invariant properties in ways that relate the data state with the synchronic groups operating over it and limit the set of synchronic groups one must consider during verification. Some of the additional safety conditions are actually rather simple and could have been isolated from the main body of the proof. Others have the effect of lengthening some of the invariants thus adding to the complexity of the proofs.

4 Dynamic synchrony

4.1 Overview of DS

The ability to alter dynamically the structure of synchronous computations provides an interesting modeling capability of potential import in specifying and reasoning about a system that combines synchronous and asynchronous features—particularly when the system may be subject to dynamic reconfiguration. However the Swarm mechanism for specifying synchronic groups, the synchrony relation, proved to be less general than expected. Dynamic Synchrony (Roman and Plun 1993) is both more general and more abstract than Swarm: actions (corresponding directly to Swarm transactions) may be enabled and disabled freely; the choice of actions to be executed synchronously is determined by the current state of the computation; the synchronization requirements are expressed as predicates; and an enabled action may belong to zero or more synchronic groups. In this section we introduce the basic computational model underlying Dynamic Synchrony (DS), with the proof system and the sample proof following in subsequent sections.

Consider a concurrent program P. In the absence of synchrony, its current state σ consists of a finite set of data objects and program actions. The data state $\sigma.d$ of P is characterized by the set of data objects currently in existence and the control state $\sigma.a$ by the set of program actions currently enabled. The data objects come from a possibly infinite universe of data objects D. The program actions come from a possibly infinite universe of program actions A. Each action denotes a nondeterministic, terminating, atomic transformation of the program state. In the case of a sequential program operating over a set of simple variables, for instance, the data state may be viewed as a set of name-value pairs and the control state consists of a single action, the statement indicated by the program counter.

As in Swarm, synchrony is defined as the coordinated execution of two or more actions of program P. The effect of executing a particular group (set) of actions is assumed to be problem-specific but finite and atomic. The entire group is treated as a single action. If individual actions are used to model assignment statements operating on simple variables, for instance, a convention must be adopted to deal with the consequence of assigning multiple distinct values to the same variable—a nondeterministic, minimum or null value are all acceptable choices. As shown later in this section, the interpretation associated with the execution of a (possibly singleton) group of actions is stated in terms of assertions (Hoare triples) about changes in the program state caused by the group's execution. DS assumes that such assertions are available for the problem at hand.

As in Swarm, a set of actions that is executed synchronously is called a synchronic group and includes only enabled actions. The composition of synchronic groups may change with each state transition even if the set of enabled actions is constant. While in Swarm every enabled action (i.e. transaction) belongs to precisely one synchronic group, in DS an enabled action may belong to several synchronic groups or to none. Furthermore, in DS the set of synchronic groups associated with a given program state can be any desired subset of the powerset of enabled actions while in Swarm they are restricted to a partition.

Given an arbitrary group of actions γ and the current state σ of a program P, we use the predicate $\Xi(\gamma, \sigma)$ to state that γ is a synchronic group (i.e. executable) in state σ. However, in DS we find it convenient to specify executability indirectly in terms of another predicate which identifies all the synchronic groups that might feasibly exist in a state σ. $\Phi(\gamma, \sigma)$, a problem specific predicate, states that the actions in γ may be executed synchronously, i.e. γ is said to be feasible but not necessarily executable since it may be outside the powerset of enabled actions. To state the requirement that all actions in γ are also enabled, we use the notation $E(\gamma, \sigma)$, i.e. $E(\gamma, \sigma) \equiv \gamma \subseteq \sigma.a$. This leads to the defining executability as $\Xi(\gamma, \sigma) \equiv \Phi(\gamma, \sigma) \wedge E(\gamma, \sigma)$. For the sake of brevity, we overload the notation by omitting the state σ whenever the state may be deduced from the context. Finally, we often use the notation α in place of $\{\ \alpha\ \}$ for sets consisting of single actions.

Starting from some valid initial state σ_0, an execution of a program P is an alternating sequence of program states and synchronic groups:

$$\sigma_0\ \gamma_0\ \sigma_1\ \gamma_1\ \sigma_2\ \gamma_2\ \sigma_3\ \gamma_3\ \sigma_4\ \cdots$$

Each subsequence of the form $\sigma\ \gamma\ \sigma'$, also called a step, corresponds to the selection of a (non-empty) synchronic group in the state σ and its execution resulting in the state

σ'. At each step, any enabled action is selected and some synchronic group to which it belongs is executed. By convention, the selection of an enabled action that does not belong to any synchronic group has no effect on the computation. A computation is considered terminated once no more synchronic groups can be selected. All executions are considered infinite by extending finite ones with pairs consisting of an empty set of actions and the final state of the finite execution. An execution is said to be fair if any continuously enabled action is eventually selected for execution. Only fair execution sequences are considered when proving a program correct. Hence, some action that is continuously enabled may never be executed, because it may not belong to any synchronic group.

All existing programming languages and models include notation and constructs that specify which actions are enabled. Flow-of-control constructs serve this purpose in imperative languages. The availability of input identifies enabled functions in dataflow languages. The mere presence of a statement indicates enablement in UNITY. The predicate E (enabled) is simply an abstraction for any such mechanism used to specify which actions are enabled in the current state.

Similarly, the predicate Φ (feasible) is an abstraction for mechanisms used to specify that two or more actions must be executed synchronously. When modeling UNITY, the predicate Φ has to capture the $\|$ construct. In the case of CSP (Hoare 1978), Φ must express the fact that matching pairs of input/output commands must execute synchronously—since an input/output command may have several acceptable matches, all possible combinations must be feasible. When modeling an SIMD machine, Φ must state that all enabled statements are executed synchronously and every processor executes the same statement. To fully appreciate the power of this model one must note that most models (with the notable exception of Swarm) do not allow the programmer to change dynamically the specification of which actions are in synchrony with which other actions. Given the fact that Φ is state-dependent, rather than merely action dependent, DS is fully capable of capturing arbitrary dynamic changes in the type of synchrony employed by programs. This, together with the fact that actions may be disabled at any time, makes DS much more general that Swarm.

4.2 Proof logic

Next we consider the DS programming logic. As in UNITY and Swarm, properties of individual actions and groups can be characterized using Hoare triples. As before, an assertion of the form

$$\{p\}\ \gamma\ \{q\}$$

states that the execution of a synchronic group γ in a state satisfying p always results in a state satisfying predicate q. Since actions that are not part of any synchronic group may be selected for execution we assume, by definition, that their execution leaves the program state unchanged. Because of the fairness requirement, if the action remains continuously enabled it will eventually be reselected for execution. An action may be selected infinitely often but still have no effect on the computation if every time it is selected the action happens not to belong to any synchronic group. While in UNITY and Swarm fair selection translates into fair execution, in DS this is not the case.

Given two predicates p and q, p **unless** q specifies that, whenever the program is in a state satisfying p but not q, any state change will result in a state still satisfying either p or q. In UNITY, this has to be proven solely for every possible statement of the program taken individually, while in Swarm over the universe of possible synchronic groups. In DS, as any group of actions can potentially be executed together, we need to prove that any subset of the universe A of actions will either maintain p or establish q. More formally, the inference rule needed to establish p **unless** q has the form:

$$\frac{\langle \forall\, \gamma : \gamma \subseteq A :: \{\ p \wedge \neg q\ \}\ \gamma\ \{\ p \vee q\ \}\rangle}{p\ \textbf{unless}\ q}$$

It should be obvious that any set of actions that is not executable in any state satisfying p, trivially preserves p. Given the **unless** relation, the definitions of **stable**, **invariant** and **constant** remain unchanged.

Given two predicates p and q, p **ensures** q is true if (1) p **unless** q holds and (2) q is eventually established. In UNITY, this is the case if there exists at least one statement that, executed in a state in which $p \wedge \neg q$ is true, always establishes q; Swarm requires that (1) some action α be enabled in the dataspace satisfying $p \wedge \neg q$, and (2) every synchronic group containing α will, if executed, establish q. In DS, we need to further expand this definition to take into account that an action can be removed before being executed and an enabled action need not be a member of any synchronic group. Thus, we define the **ensures** relation as follows: Given two predicates p and q, p **ensures** q is true if (1) p **unless** q holds, (2) there exists an action α that remains part of some synchronic group until q is established and (3) any executable synchronic group containing α establishes q upon execution in a state satisfying $p \wedge \neg q$:

$$\frac{\begin{array}{r} p\ \textbf{unless}\ q, \\ \langle \exists\, \alpha : \alpha \in A :: (p \wedge \neg q \Rightarrow \langle \exists\, \gamma : \gamma \subseteq A \wedge \alpha \in \gamma :: \Xi(\gamma)\rangle) \wedge \\ \langle \forall\, \gamma : \gamma \subseteq A \wedge \alpha \in \gamma \wedge \Xi(\gamma) :: \{p \wedge \neg q\}\ \gamma\ \{q\}\rangle\rangle \end{array}}{p\ \textbf{ensures}\ q}$$

The definition of the **leads-to** relation remains unchanged.

4.3 Example proof

Since no concrete syntax has been developed yet for DS, the DS program discussed in this section is given using the basic Swarm notation except that the query and manipulation of synchrony relation entries are disallowed and a section that defines the feasibility function Φ is added. The resulting program is shown in Fig. 4. While the transactions $Sum(k, j)$ no longer create and delete synchrony relation entries, the synchronic groups are the same as in the Swarm version of the program. All the transactions participating in the phase j of the summation form a feasible group γ_j and each feasible group is enabled precisely when the computation reaches that phase. In the remainder of this section we consider the implications of these program changes on its correctness proof.

If we can show that corresponding data states in the Swarm and the DS programs have the same synchronic groups, the correctness of the DS program follows from the proof in section 3. Formally, we have to revisit the Property 7 (Structure Invariant) and to show that it holds in the DS program. This condition can be restated as follows:

program $DS\_Sum(N, A : [\exists p : p \geq 0 :: N = 2^p], A(i : 1 \leq i \leq N))$
tuple types
$[i, s : 1 \leq i \leq N :: x(i, s)]$
transaction types
$[k, j : 1 \leq k \leq N,\ 1 \leq j < N ::$
$Sum(k, j)\ \equiv$
$v1, v2 : x(k-j, v1)\dagger,\ x(k, v2)\dagger\ \rightarrow\ x(k, v1+v2)$
$\|\quad j < N,\ k \,\mathbf{mod}\, (j * 4) = 0\ \rightarrow\ Sum(k, j * 2)$
$]$
initialization
$[i : 1 \leq i \leq N :: x(i, A(i))]$
$[k : 1 \leq k \leq N, k \,\mathbf{mod}\, 2 = 0 :: Sum(k, 1)]$
synchronization
$\gamma_j\quad \equiv\ [\mathbf{set}\ k : 1 \leq k \leq N,\ k \,\mathbf{mod}\, (2 * j) = 0 :: Sum(k, j)]$
$\Phi(\gamma)\quad \equiv\ [\exists\, j :: \gamma = \gamma_j]$
end

Figure 4: Array summation in DS using a Swarm-like notation

Property 9 (Modified Structure Invariant)

invariant
$pow2(N)\ \wedge\ pow2(W_x)\ \wedge$
$(\, x(i, v)\ \equiv\ node(i, W_x) \wedge v = sum_A(i - W_x, i)\,)\ \wedge$
$(\, Sum(i, j)\ \equiv\ node(i, 2 * j) \wedge j = W_x\,)\ \wedge$
$(\, \Xi(\gamma_j)\ \equiv\ j = W_x\,)$

Proof of the Modified Structure Invariant: In the initial state there is only one synchronic group, γ_1, and the metric W_x assumes the value 1. The synchronic group γ_1 contains a *Sum* transaction for each even position in the array. For any j less than $N/2$, the execution of the synchronic group γ_j leads to the creation of all and only transactions that are part of γ_{2*j}, itself a feasible group. For j equal to $N/2$, the execution of the synchronic group γ_j leads to an empty transaction space, i.e. no synchronic group to continue the execution. γ_N is empty by definition. The impact on array elements and transactions is the same as in the Swarm program. ∎

The changes to the section 3 proof are minimal because the programs appearing in Fig. 3 and Fig. 4 are operationally similar. A small change to the program in Fig. 4, however, provides a more dramatic illustration of the expressive power of DS. Rather than create all the *Sum* transactions in a single step, we introduce an intermediary *MakeSum* transaction that simply forces the creation of the *Sum* transactions to be asynchronous. The synchronization section is also modified to ensure that the only feasible transaction

program $DS\_Sum\_Variant$
$(N, A : [\exists p : p \geq 0 :: N = 2^p], A(i : 1 \leq i \leq N))$
tuple types
$[i, s : 1 \leq i \leq N :: x(i, s)]$
transaction types
$[k, j : 0 \leq k \leq N,\ 1 \leq j < N ::$
$Sum(k, j) \equiv$
$v1, v2 : x(k-j, v1)\dagger,\ x(k, v2)\dagger \rightarrow x(k, v1+v2)$
$\|\quad j < N,\ k \bmod (j * 4) = 0 \rightarrow MakeSum(k, j * 2)$

$MakeSum(k, j) \equiv$
$\mathbf{true} \rightarrow Sum(k, j)$
$]$
initialization
$[i : 1 \leq i \leq N :: x(i, A(i))]$
$[k : 1 \leq k \leq N,\ k \bmod 2 = 0 :: MakeSum(k, 1)]$
synchronization
$\gamma_j \equiv [\mathbf{set}\ k : 1 \leq k \leq N,\ k \bmod (2 * j) = 0 :: Sum(k, j)]$
$\Phi(\gamma) \equiv [\exists\, j :: \gamma = \gamma_j] \vee [\exists\, k, j :: \gamma = [\mathbf{set} :: MakeSum(k, j)]\ ]$
end

Figure 5: Partially asynchronous array summation in DS

groups are either individual *MakeSum* transactions or complete sets of *Sum* transactions corresponding to instances of γ_j. Individual *Sum* transactions may be present in the dataspace but even if executed they have no effect until the full set is created. This modified program is shown in Fig. 5.

This time both Property 7 (Structure Invariant) and Property 8 (Sum Step) are affected by this program change. The Structure Invariant must accommodate the fact that a *MakeSum* transaction may be substituting for a corresponding *Sum* transaction and that γ_j is not executable as long as there are still *MakeSum* transactions around. In the Sum Step the ensures changes to a leads-to.

Property 10 (Revisited Structure Invariant)

invariant
$pow2(N) \wedge pow2(W_x) \wedge$
$(x(i, v) \equiv node(i, W_x) \wedge v = sum_A(i-W_x, i)) \wedge$
$(MakeSum(i, j) \Rightarrow \neg Sum(i, j)) \wedge$
$(Sum(i, j) \vee MakeSum(i, j) \equiv node(i, 2 * j) \wedge j = W_x) \wedge$
$(\Xi(\gamma_j) \equiv j = W_x \wedge \langle \forall i :: \neg MakeSum(i, j) \rangle)$

Property 11 (Revisited Sum Step) $W_x = k < N \;\longmapsto\; W_x = 2 * k$

The proof of Property 10 is not fundamentally different from the earlier version. The proof of Property 11 requires one to show that the number of *MakeSum* transactions decreases to zero (due to the fact that they are the only transactions that can execute) at which point the appropriate γ_j is formed and the summation step actually takes place. We omit the actual proof.

5 Conclusions

The starting point for this work was UNITY and its proof logic. Swarm and DS represent two progressively more radical and dynamic departures from the basic UNITY model. Nevertheless, most of the structure of the UNITY proof logic carried over. Only the proof obligations for the **unless** and **ensures** relations have been affected by the new features. In retrospect, this can be easily explained by the fact that these relations are the only ones to be defined in terms of properties of individual program actions. Even though Swarm and DS provide novel mechanisms for determining what an action is, they preserve the basic notion of atomic action. For static problems, the use of Swarm or DS incurs only minor increases in verification complexity. For problems that involve dynamic data and action creation, Swarm provides a convenient notation and the simplicity of the UNITY-like proof logic. DS does the same thing for problems that involve reasoning about synchrony.

Because the three models share a common proof logic, program derivation techniques based on specification refinement often need not worry about which of the three models is ultimately used to describe the resulting program. This may not be the case in the future as program derivation techniques that exploit features specific to Swarm and DS are developed. To date, Swarm has been used already in program derivation efforts that involve a combination of UNITY-style specification refinements followed by Swarm-specific program refinements targeted to achieving executability on specific architectures. DS has not been employed in any research on formal derivation. Its most interesting application to date has been in showing that it can be useful in constructing UNITY-like assertional proofs for models completely unrelated to UNITY. The key here has been the DS ability to formalize their particular synchronization mechanisms.

Acknowledgements

This work was supported, in part, by the National Science Foundation (USA) under grants CCR-9210342 (first author) and CCR-9015677 (second and third authors).

References

Andrews, G. R. and Reitman, R. P. (1980). Axiomatic approach to information flow in programs, *ACM Transactions on Programming Languages and Systems* **2**(1): 56–76.

Apt, K. R., Francez, N. and de Roever, W. P. (1980). A proof system for Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems* **2**(3): 359–385.

Brownston, L., Farrell, R., Kant, E. and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, Massachusetts, USA.

Chandy, K. M. and Misra, J. (1988). *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, USA.

Cunningham, H. C. and Roman, G.-C. (1990). A UNITY-style programming logic for shared dataspace programs, *IEEE Transactions on Parallel and Distributed Systems* **1**(3): 365–376.

Dijkstra, E. W. (1976). *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Floyd, R. (1967). Assigning meaning to programs, *in* J. T. Schwartz (ed.), *Mathematical Aspects of Computer Science*, American Mathematical Society, Providence, Rhode Island, USA, pp. 19–32.

Francez, N. (1986). *Fairness*, Springer-Verlag, New York.

Gamble, R. F., Roman, G.-C., Ball, W. E. and Cunningham, H. C. (in press). Applying formal verification methods to rule-based programs, *International Journal of Expert Systems Research and Applications*.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming, *Communications of the ACM* **12**(10): 576–580.

Hoare, C. A. R. (1978). Communicating sequential processes, *Communications of the ACM* **21**(8): 666–677.

Lamport, L. (1977). Proving the correctness of multiprocess programs, *IEEE Transactions on Software Engineering* **3**(2): 125–143.

Lamport, L. (1980). The Hoare logic of concurrent programming, *Acta Informatica* **14**(1): 21–37.

Lynch, N. A. and Tuttle, M. R. (1989). An introduction to input/output automata, *CWI Quarterly* **2**(3): 219–246.

Manna, Z. and Pnueli, A. (1974). Axiomatic approach to total correctness of programs, *Acta Informatica* **3**: 243–264.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York.

Milne, G. and Milner, R. (1979). Concurrent processes and their syntax, *Journal of the ACM* **26**(2): 302–321.

Misra, J. (1990). Soundness of the substitution axiom, Notes on UNITY 14–90, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, USA.

Morris, J. M. (1989). Laws of data refinement, *Acta Informatica* **26**: 287–308.

Owicki, S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach, *Communications of the ACM* **19**(5): 279–285.

Roman, G.-C. and Cunningham, H. C. (1990). Mixed programming metaphors in a shared dataspace model of concurrency, *IEEE Transactions on Software Engineering* **16**(12): 1361–1373.

Roman, G.-C. and Cunningham, H. C. (1992). Reasoning about synchronic groups, *in* J. P. Banâtre and D. Le Métayer (eds), *Research Directions in High-level Parallel Programming Languages,* LNCS #574, Springer-Verlag, New York, pp. 21–38.

Roman, G.-C. and Plun, J. (1993). Reasoning about synchrony illustrated on three models of concurrency, *Technical Report WUCS–93–4*, Department of Computer Science, Washington University, St. Louis, Missouri, USA.

Sanders, B. (1991). Eliminating the substitution axiom from UNITY logic, *Formal Aspects of Computing* **3**(2): 189–205.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*, MIT Press, Cambridge, Massachusetts, USA.