Report Number: WUCS-93-4

1993-01-01

Reasoning about Synchrony Illustrated on Three Models of Concurrency

Gruia-Catalin Roman and Jerome Plun

This paper presents a model of concurrency (Dynamic Synchrony) whose distinctive feature is a novel formal treatment of synchronization. Synchrony is defined as the coordinated execution of two or more actions. The dynamic aspect comes from the fact that the definition of which actions must be executed synchronously can change freely during the execution of the program. This unique modeling capability comes with a UNITY-stype assertional logic that can be applied to program verification and derivation. This paper shows that the proposed proof logic can be used to verify programs expressed using other models of foncurrency without having to... **Read complete abstract on page 2.**

Reasoning about Synchrony Illustrated on Three Models of Concurrency

Gruia-Catalin Roman and Jerome Plun

Complete Abstract:

This paper presents a model of concurrency (Dynamic Synchrony) whose distinctive feature is a novel formal treatment of synchronization. Synchrony is defined as the coordinated execution of two or more actions. The dynamic aspect comes from the fact that the definition of which actions must be executed synchronously can change freely during the execution of the program. This unique modeling capability comes with a UNITY-stype assertional logic that can be applied to program verification and derivation. This paper shows that the proposed proof logic can be used to verify programs expressed using other models of foncurrency without having to translate them to our notation. This capability is illustrated by verifying three versions of a parallel array summation problem, each written using a different model and notation - Swarm, Concurrent Processes, and Input/Output Automata. The new model makes UNITY-stype proofs feasible for a broad range of models of concurrency regardless of the way they handle synchronization and even if they lack an associated proof logic.

Washington

WASHINGTON • UNIVERSITY • IN • ST • LOUIS

School of Engineering & Applied Science

Reasoning about Synchrony Illustrated on Three Models of Concurrency

Gruia-Catalin Roman
Jerome Plun

WUCS-93-04

January 1993

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

Printed 1/25/93

Abstract

This paper presents a model of concurrency *(Dynamic Synchrony)* whose distinctive feature is a novel formal treatment of synchronization. Synchrony is defined as the coordinated execution of two or more actions. The dynamic aspect comes from the fact that the definition of which actions must be executed synchronously can change freely during the execution of the program. This unique modeling capability comes with a UNITY-style assertional logic that can be applied to program verification and derivation. This paper shows that the proposed proof logic can be used to verify programs expressed using other models of concurrency without having to translate them to our notation. This capability is illustrated by verifying three versions of a parallel array summation problem, each written using a different model and notation—*Swarm, Concurrent Processes,* and *Input/Output Automata.* The new model makes UNITY-style proofs feasible for a broad range of models of concurrency regardless of the way they handle synchronization and even if they lack an associated proof logic.

Correspondence: All communications regarding this paper should be addressed to

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

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

roman@swarm.wustl.edu

1. Introduction

Reliability, a growing concern in the software development arena, has lead to increased dependence on formal proofs of critical components and algorithms. The use of assertional-style proof techniques was formally introduced for sequential algorithms by Floyd [6]. This work was further developed by Hoare [10] and Manna and Pnueli [16] 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 [5]) and data refinement (Gries [8]).

The correctness of concurrent programs was first tackled by Owicki and Gries [19] for proving partial correctness, mutual exclusion, and absence of deadlock. In related but independent work, Lamport [12] describes the use of control predicates to prove interference freedom in programs. This paper also introduced the terms *safety* and *liveness*. Later on, Apt, Francez, and De Roever [2] extended Hoare's Communicating Sequential Processes (CSP) [11] with an axiomatic proof system, and Andrews and Reitman [1] proposed an axiomatic proof system based on the information flow in a program. Generalized Hoare Logic was developed by Lamport [13] to verify safety properties of concurrent programs by using an invariant to describe a program.

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 arsenal of conceptual tools. Our experience indicates that the UNITY logic [3] 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 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 restricted case of temporal logic [15] . 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 permissible state transitions while *progress* properties specify which state transitions must occur.

Building upon UNITY, the Swarm model [20] and its proof system [4, 21] have extended the applicability of assertional-style proofs to content-addressable data access, to dynamic data and actions, and to dynamic forms of synchrony between actions. This allowed the use of assertional-style proofs in rule-based programming [7], made it possible to reason about programs involving multiple computing paradigms (such as shared-variable, message

passing, and rule-based), and opened the possibility of verifying software which executes on networks consisting of a mixture of synchronous and asynchronous machines.

The fact that the UNITY logic was amenable to extensions that represent such a major departure from UNITY itself begs the question of whether it could be applied to an even broader set of models and programming languages. One first step in this direction is to develop a proof logic which enable us to construct UNITY-like assertional proofs for programs expressed in other models of concurrency. In this paper, we propose such a framework, named the dynamic synchrony (DS) model, as an extension of the Swarm model and show its usefulness by proving the correctness of an array summation program expressed in three different models of concurrency: Swarm, Concurrent Processes (CP) [17], and Input/Output Automata (IOA) [14]. It is important to note that neither CP nor I/O Automata have assertional-style proof logics. We believe that the same framework will later prove usable in conjunction with actual programming languages.

Although the three models considered here employ radically different forms of synchronization, the verification is carried out by the same proof logic. This is because in our framework synchronization rules are explicitly 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 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.

In the following, Section 2 describes the framework and its proof system. Section 3 contains the definition of the array summation problem and its principal proof obligations while Sections 4, 5, and 6 contain the representation of the program and its proofs for Swarm, Concurrent Processes, and Input/Output Automata, respectively. Section 7 presents conclusions and indicates the direction of future research.

2. A Framework for Synchrony

Consider a concurrent program P. In the absence of synchrony, its current state σ is fully determined by its data and control states. Given some possibly infinite universe of data objects D and atomic program actions A, 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. 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.

We define synchrony as the coordinated execution of two or more actions of program P. Any single action can always be seen as coordinating its execution with itself. Two distinct actions that coordinate their execution are said to be in synchrony with each other. We require actions which are in synchrony which each other to be executed together as a group. Given an arbitrary group (set) of actions γ and the current state σ of a program P, γ may be executed if and only if all its actions are enabled and in synchrony with each other. A set of actions which is executable is called a synchronic group. We use the predicate $\Xi(\gamma, \sigma)$ to state that a group of actions γ is *executable* in state σ, $E(\gamma, \sigma)$ to state that all actions in γ are *enabled*, and $\Phi(\gamma, \sigma)$ to state that the actions in γ must be executed synchronously, i.e., γ is said to be *feasible*. $\Phi(\gamma, \sigma)$ is problem specific, $E(\gamma, \sigma) \equiv \gamma \sqsubseteq \sigma.a$, and $\Xi(\gamma, \sigma) \equiv \Phi(\gamma, \sigma) \wedge E(\gamma, \sigma)$. 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:

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

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 which 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.

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 of more actions must be executed synchronously. When modeling UNITY, the predicate Φ has to capture the "||" construct. In the case of CSP[11], Φ must express the fact that matching pairs of input/output commands must execute synchronously. 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, our framework is fully capable of capturing dynamic changes in the type of synchrony employed by programs. In Swarm, for instance, the programmer has the ability to bring in and out of synchrony any actions one may desire, i.e., the same two actions may be executed either synchronously or asynchronously under direct programmer control. Even this rather unique feature of Swarm (together with all its idiosyncrasies) can be easily captured by a proper definition of Φ. We will show that the dynamic synchrony model is much more general that Swarm.

Next we extend this framework with a proof system based on the assertional logic of Swarm[4, 21] and indirectly on that of UNITY [3]. We use, for the most part, the notational conventions of UNITY. For the sake of clarity, free variables appearing in properties and inference rules are assumed to be universally quantified by implication.

The meaning of an action α is specified by a Hoare triple [10]: $\{ p \} \alpha \{ q \}$ where p and q are predicates on the computation state before and after executing α, respectively. Informally, this assertion is true if and only if forcing the execution of action α (whether executable or not) in a state satisfying p could result in a state satisfying predicate q. By extension, the meaning of a set of actions $\gamma = \{ \alpha_1, \ldots, \alpha_n \}$ can be specified by a "Hoare" triple $\{ p \} \gamma \{ q \}$. These formulas reflect the potential effect of executing an action or a group of actions, regardless of whether or not these actions are actually executable or even enabled in the current state.

A program, however, does not execute arbitrary actions but synchronic groups, i.e., sets of actions which are both enabled and feasible. The inference rule (IR1) states that an executable synchronic group does in fact change the program state:

(IR1)
$$\frac{\{ p \} \gamma \{ q \}, \; p \Rightarrow \Xi(\gamma)}{\{ p \} \gamma \{ q \}}$$

Of course, a non-executable synchronic group can not change the state of the program:

(IR2)
$$\frac{p \Rightarrow \neg(\Xi(\gamma))}{\{ p \} \gamma \{ p \}}$$

As in UNITY, all safety properties of a program are specified in terms of the **unless** relation. 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 the UNITY model, this has to be proven solely for every possible statement of the program taken individually. 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:

(IR3)
$$\frac{\langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} :: \{\, p \wedge \neg q \,\}\ \gamma\ \{\, p \vee q \,\}\, \rangle}{\mathbf{p\ unless\ q}}$$

It should be obvious that any set of actions which is not executable in any state satisfying p, trivially preserves p. This definition mirrors the one in the Swarm model if γ is restricted to the universe of synchronic groups rather that the powerset of **A**. Based on the **unless** relation, we define the **stable** and invariant (**inv.**) relations as in UNITY:

$$\mathbf{stable}\ p \quad \equiv p\ \mathbf{unless}\ \mathrm{false}$$

$$\mathbf{invariant}\ p \equiv \mathrm{INIT} \Rightarrow p \wedge \mathbf{stable}\ p$$

where INIT is a predicate describing the initial state of the computation.

The most basic kind of progress property is expressed using the **ensures** relation. 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 one statement which, executed in a state in which $p \wedge \neg q$ is true, establishes q. In constrast, due to its dynamic nature, 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 in DS 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 which 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 statisfying $p \wedge \neg q$:

(IR4)
$$\frac{\begin{array}{l}\mathbf{p\ unless\ q},\\ \langle\, \exists\, \alpha : \alpha \in \mathbf{A} :: \quad (p \wedge \neg q \Rightarrow \langle\, \exists\, \gamma : \gamma \subseteq \mathbf{A} \wedge \alpha \in \gamma :: \Xi(\gamma)\, \rangle) \wedge \\ \qquad\qquad \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \alpha \in \gamma \wedge \Xi(\gamma) :: \{\, p \wedge \neg q \,\}\ \gamma\ \{\, q \,\}\, \rangle\, \rangle\end{array}}{\mathbf{p\ ensures\ q}}$$

The requirement that p must hold while q is not true is relaxed in the definition of the *leads-to* relation (represented, as in UNITY, by the symbol $\mapsto$). The property $p \mapsto q$ holds if and only if it can be derived from a finite number of applications of the following inference rules:

1) ensures $\dfrac{p \textbf{ ensures } q}{p \mapsto q}$

2) transitivity $\dfrac{p \mapsto q \ , \ q \mapsto r}{p \mapsto r}$

3) disjunction $\dfrac{\langle \forall \ m : m \in \mathrm{W} :: p(m) \mapsto q \rangle}{\langle \exists \ m : m \in \mathrm{W} :: p(m) \rangle \mapsto q}$

3. An example: array summation

In the following sections, we use the DS logic to prove correct three versions of a parallel array summation program expressed, respectively, in Swarm, CP, and IOA. Because the three models employ distinct synchronization strategies, the three programs shape the computation in distinct ways. The summation of the N element array A is accomplished by the successive computation of partial sums (adapted from the one given in [9]). For simplicity of presentation we assume that N is a power of 2. In all cases the summation process may be perceived as following a tree structure with the leaf nodes corresponding to the initial array values and the internal nodes to partial sums (see Fig. 1). The computation advances by computing new partial sums until the root contains the sum over all array values. For reasons of convenience, all array values or partial sums already subsumed by nodes further in the tree are discarded or ignored, while partial sums not yet computed are treated as undefined.

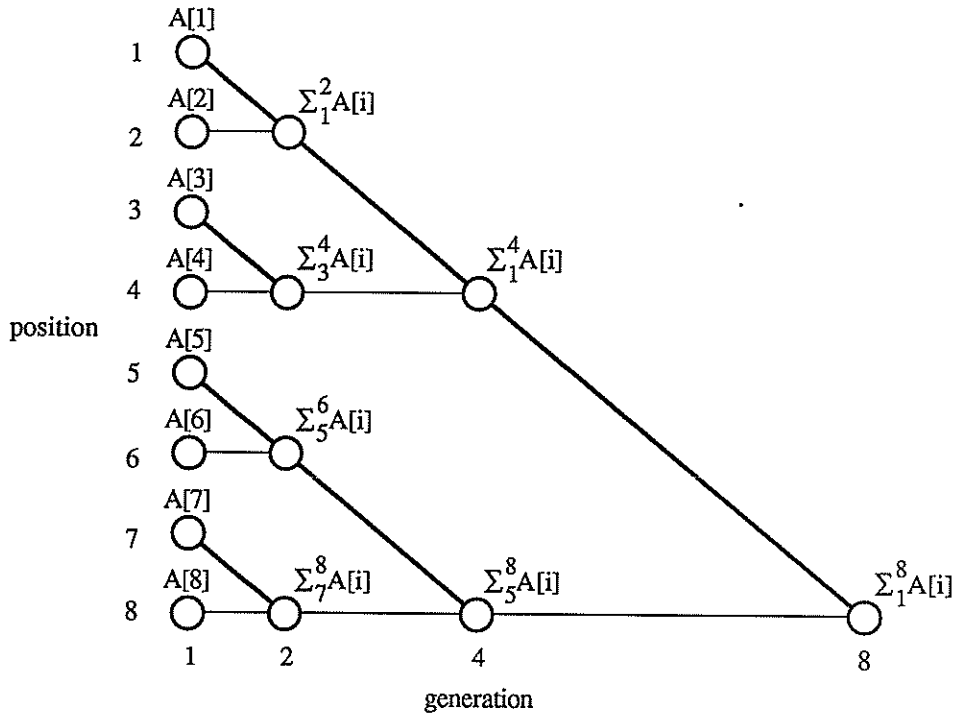

Fig. 1. The summation tree for N = 8

Let T denote the set of nodes in the summation tree. The location of each node n in the tree is defined in terms of a generation *gen(n)*—powers of 2—and a position within the generation *pos(n)*, as shown in Fig. 1. A value *val(n)* is associated with each node. This value is either some value in the array **A**, some partial sum, or the special symbol $\perp$ indicating that the value of the node is undefined or is to be ignored. The predicate *leaf(n)* is true if the node n is a leaf of T. For a non-leaf node n, *left(n)* and *right(n)* represent the left and right child, respectively. Finally, the function *root(T)* returns the root node of T, and *tree(n)* the subtree rooted at node n. The formal definitions of *gen(n)*, *pos(n)*, and *tree(n)* are as follows:

$$\text{gen(n)} = \begin{cases} 1 & \text{if leaf(n)} \\ 2\text{gen(left(n))} & \text{if } \neg\text{leaf(n)} \end{cases}$$

$$\text{pos(n)} = \begin{cases} N & n = \text{root(T)} \\ \text{pos(m)-gen(m)} & n = \text{left(m)} \\ \text{pos(m)} & n = \text{right(m)} \end{cases}$$

$$\text{tree(n)} = \begin{cases} \{\ n\ \} & \text{leaf(n)} \\ \text{tree(left(n))} \cup \{\ n\ \} \cup \text{tree(right(n))} & \neg\text{leaf(n)} \end{cases}$$

From these definitions, we have the following properties[1]:

$$1 \le \text{gen(n)} \le N \wedge \langle\, \exists\, j : j \ge 0 :: \text{gen(n)} = 2^j \,\rangle$$

$$1 \le \text{pos(n)} \le N \wedge \text{pos(n) mod gen(n)} = 0$$

$$n = \text{root(T)} \Leftrightarrow \text{gen(n)} = \text{pos(n)} = N$$

$$m = \text{left(n)} \Leftrightarrow \neg\text{leaf(n)} \wedge \text{gen(m)} = \frac{\text{gen(n)}}{2} \wedge \text{pos(m)=pos(n)–gen(n)}$$

$$m = \text{right(n)} \Leftrightarrow \neg\text{leaf(n)} \wedge \text{gen(m)} = \frac{\text{gen(n)}}{2} \wedge \text{pos(m)} = \text{pos(n)}$$

The basic specification of the array summation problem is given by four properties which state that 1) initially only leaf nodes have defined value, the array element corresponding to the position of the node, 2) the sum

---

[1] The three-part notation *⟨ op quantified_variables : range :: expression ⟩* used throughout the text is defined as follows. The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which ***op*** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for ***op***, e.g., *true* when ***op*** is $\forall$, 0 when ***op*** is Σ, etc.

of the values stored in the computation tree, excluding the ignored or undefined nodes, is constant and equal to the sum of the array values, 3) the value at the root node of T eventually becomes the sum of the array values and 4) once the value of the root is equal to the sum, it remains such. Formally, they are captured by the assertions:

(P0) $\text{INIT} \Rightarrow \langle\, \forall\, n : n \in T :: (\text{leaf}(n) \Rightarrow \text{val}(n) = A[\text{pos}(n)]) \wedge (\neg\text{leaf}(n) \Rightarrow \text{val}(n) = \perp) \,\rangle$

(P1) **inv.** $\text{Sum}(T) = \text{SumA}$

(P2) $\text{INIT} \mapsto \text{val}(\text{root}(T)) = \text{SumA}$

(P3) **stable** $\text{val}(\text{root}(T)) = \text{SumA}$

where *Sum(T)* is the sum of the nodes in tree T whose value is not $\perp$, and *SumA* is the sum of the elements of the array A, i.e.,

$$\text{SumA} = \langle\, \sum i : 1 \le i \le N :: A[i] \,\rangle$$

$$\text{Sum}(t) = \langle\, \sum n : n \in t \wedge \text{val}(n) \neq \perp :: \text{val}(n) \,\rangle$$

To prove the progrress conditions of the individual programs, we need to define a well founded metric and to show that the metric decreases during the computation, eventually reaching a minimum value which marks the completion of the summation process. We chose as our metric to be:

$$\mathcal{M} = \langle\, \sum n : n \in T \wedge \text{val}(n) \neq \perp :: N - \text{gen}(n) \,\rangle$$

Since *gen(n)* is at most N, the metric is well founded. Moreover, one can easily show, based on the definition of $\mathcal{M}$ and property (P0) that:

(D1) $\text{INIT} \Rightarrow \mathcal{M} = N*(N-1)$

(D2) **inv.** $\mathcal{M} = 0 \Rightarrow \text{val}(\text{root}(T)) = \text{SumA}$

We take advantage of (D2) and refine (P3) to be:

(P3.1) **stable** $\mathcal{M} = 0$

As mentioned before, progress will be proven by showing that the metric decreases at each step and, eventually, reaches zero. This suggests the following refinement of (P2):

(P2.1) $\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$

¶ **Proof:** (P2.2) is a refinement of (P2)

$$\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$$

$\Rightarrow$ [$\mathcal{M} = 0 \mapsto \mathcal{M} = 0$, disjunction rule for $\mapsto$]

$$(\mathcal{M} = k \wedge k > 0) \vee \mathcal{M} = 0 \mapsto \mathcal{M} < k \vee \mathcal{M} = 0$$

$\Rightarrow$ [calculus]

$\mathcal{M} = k \mapsto \mathcal{M} < k \vee \mathcal{M} = 0$

$\Rightarrow$ [induction rule on $\mapsto$]

$\mathcal{M} = k \mapsto \mathcal{M} = 0$

$\Rightarrow$ [INIT $\mapsto \mathcal{M} = N*(N-1)$ from (D1), $\mathcal{M} = 0 \mapsto$ val(root(T)) = SumA from (D2)]

(P2) □

It is this refined specification that will be used as the correctness criterion for the programs introduced in the following sections:

(P0) $\text{INIT} \Rightarrow \langle \forall n : n \in T :: (\text{leaf}(n) \wedge \text{val}(n) = A[\text{pos}(n)]) \vee (\neg\text{leaf}(n) \wedge \text{val}(n) = \perp) \rangle$

(P1) **inv.** Sum(T) = SumA

(P2.1) $\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$

(P3.1) **stable** $\mathcal{M} = 0$

4. Swarm

4.1. The model

The Swarm model [20] is based on the concept of a *dataspace*, a set of data items (data *tuples*) and enabled actions (*transactions*). A transaction is composed of subtransactions, each consisting of a query over the content of the dataspace and a set of modifications which are applied to the dataspace if the query succeeds. A transaction can only be removed from the dataspace if selected for execution. A data tuple (or transaction) present in the dataspace is an instance of some data tuple (transaction) type and assumes the form *name(values)*. Since the dataspace is a set, there are no duplicates of data tuples or transactions instances. Synchronization between transactions is expressed using a third component of the dataspace, the synchrony relation. This symmetric, irreflexive relation is specified by the insertion in the dataspace of entries of the form '$t_1 \sim t_2$', where t_1 and t_2 are possible transaction instances. These entries are part of the program state and, as such, can be freely created, queried, and removed by transactions. A Swarm *synchronic group* is a group of transaction related under the reflexive transitive closure of the synchrony relation. When a transaction t is selected for execution, all enabled transactions related to t are also selected and executed. The execution of a synchronic group involves two phases: first, the queries of all the subtransactions are evaluated; second, for every subtransaction having a successful query, the appropriate modifications are applied to the dataspace, with all deletions performed before any insertions.

program ArraySumSynch(N, A: ⟨∃ p: p ≥ 0 :: N = 2^p⟩, A[i: 1≤ i ≤ N])

tuple types

The nodes of the tree are represented by data tuples d(gen, p, v)*, where* gen *is the generation of the node,* p *is the position within the generation, and* v *is the value associated with the node. Only defined nodes* ($v \neq \perp$) *are present in the dataspace*

⟨ gen, p, v: 1 ≤ gen ≤ N, 1 ≤ p ≤ N :: d(gen, p, v) ⟩

transaction types

⟨ gen, p: 1 < gen ≤ N, 0 ≤ p ≤ N ::

The actions are represented by Sum(gen, p) *transactions. The role of a* Sum(gen, p) *transaction is to compute the value of the node at generation* gen *and position* p.

Sum(gen, p) ≡

The value of a node is computed by adding the values of its children nodes. The children are removed from the dataspace.

v_1, v_2: $d(\frac{gen}{2}, p-\frac{gen}{2}, v_1), d(\frac{gen}{2}, p, v_2) \rightarrow d(gen, p, v_1+v_2), d(\frac{gen}{2}, p-\frac{gen}{2}, v_1)\dagger, d(\frac{gen}{2}, p, v_2)\dagger$

Upon execution, this transaction is deleted. If the corresponding node is not the root of the tree and a node having the same position appears in the next generation, its associated transaction is created and is synchronized through a common "dummy" transaction to all tranasctions needed to compute the next generation.

‖ gen < N, p mod (2*gen) = 0 → Sum(2*gen, p), Sum(2*gen, p)~Sum(2*gen, 0)

⟩

initialization

Initially, only the leaf nodes are present in the dataspace. They contain the values from the array A

⟨ p : 1 ≤ p ≤ N :: d(1, p, A[p]) ⟩

For every even position, a transaction is created to compute the value of the node at that position and it is synchronized with a common "anchor" transaction (which, in this case, is never enabled but only serves to easily specify a synchronic group). As a result, all the nodes in the second generation will have their value computed at once.

⟨ p: 1 ≤ p ≤ N, p mod 2 = 0 :: Sum(2, p); Sum(2, p) ~ Sum(2, 0) ⟩

end

Fig. 2. Parallel array summation in Swarm

Fig. 2 shows a Swarm version of the Array Summation problem. The nodes in the computation tree are represented by *d(gen, p, v)* tuples, where *gen*, *p*, and *v* correspond to the generation, the position, and the value of the node, respectively. The computation is performed by *Sum(gen, p)* transactions. Each *Sum(gen, p)* transaction is "responsible" for creating the corresponding *d(gen, p, v)* tuple. The synchronization is such that all the transactions

creating nodes in a given generation belong to the same synchronic group. As a result, the values of all nodes equidistant from the root, i.e., having the same generation number, are computed during the same step.

4.2. Correctness Proof

Given the fact that in the Swarm program nodes having defined values are represented by tuples of the form *d(gen, p, value)* while those whose values are undefined are absent from the dataspace, the following coupling invariant relates the Swarm data representation to the *val* function appearing in the specifications:

$$\text{val(n)} = \text{v} \equiv (\text{v} \neq \perp \wedge \text{d(gen(n), pos(n), v)}) \vee (\text{v} = \perp \wedge \neg \langle \exists \text{ v} : \text{v} \neq \perp :: \text{d(gen(n), pos(n), v)} \rangle)$$

In order to verify this program, we show 1) that the computation of the partial sums is performed one generation at a time, 2) only one generation of nodes is ever present in the dataspace, and 3) at most one synchronic group is enabled—the one computing the next generation of nodes. In the remainder of this section, we will use the predicate $\mathcal{S}(g)$ to indicate that the values of the nodes of generation g are present in the dataspace and all other node values are undefined, and the set γ_g to represent the synchronic group that computes the nodes of the generation $2g$. The formal definitions of $\mathcal{S}(g)$ and γ_g are:

(SD1) $\quad \mathcal{S}(\text{g}) \equiv \langle \forall \text{ n} : \text{n} \in \text{T} :: \text{gen(n)} = \text{g} \Leftrightarrow \text{val(n)} \neq \perp \rangle$

(SD2) $\quad \gamma_\text{g} \equiv \langle \textbf{set p} : 1 \leq \text{p} \leq \text{N} \wedge \text{p mod 2g} = 0 :: \text{Sum(2g, p)} \rangle$

We will show later on that at most one synchronic group is ever enabled, that this enabled synchronic group corresponds to some γ_g, and that the set of data tuples in the dataspace at any time is represented by some instance of predicate $\mathcal{S}$. As a result, we will only define the meaning of executing a synchronic group γ_g in a state for which $\mathcal{S}(\text{g})$ is true and no other synchronic groups are enabled. Based on the description of a *Sum(gen, p)* transaction, the effect of executing γ_g is fourfold: 1) all data items at generation g are set to $\perp$ (removed from the dataspace), 2) all the data items at generation $2g$ are set to the sum of the values of their respective children, 3) all transactions Sum($2g$, p) are deleted (implicitely by being executed), and 4) when $g < N$ all transactions necessary to create the data items at generation 2g are inserted in the dataspace and synchronized with a common transaction *Sum*($4*g$, 0), thus enabling $\gamma_{2\text{g}}$ (while leaving all other possible synchronic groups disabled). This is formally expressed as:

(SD3)
$$[\ \mathcal{S}(\text{g}) \wedge \Xi(\gamma_\text{g}) \wedge \langle \forall\ \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_\text{g} :: \neg\Xi(\gamma) \rangle\ ]$$

$$\gamma_\text{g}$$

$$[\ \mathcal{S}(2\text{g}) \wedge \neg\Xi(\gamma_\text{g}) \wedge (\Xi(\gamma_{2\text{g}}) \Leftrightarrow\ 2\text{g} < \text{N}) \wedge \langle \forall\ \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_{2\text{g}} :: \neg\Xi(\gamma) \rangle\ ]$$

In order to prove that the Swarm program satisfies the earlier specifications, we first establish four properties relating $\mathcal{S}(\text{g})$ and γ_g to the behavior of the program. These properties are later used to verify the

correctness of the program. The first two properties, (SI1) and (SI2), specify what transactions are enabled based on the generation g of nodes whose values are defined. If this generation does not include the root node, there is exactly one synchronic group, γ_g, in the dataspace. If the generation includes the root, there are no enabled transactions. The other two properties constrain what data tuples can be present in a dataspace. (SI3) indicates that the values of a generation remain in the dataspace at least until the next generation is computed and that the next generation will be computed, if possible, while (SI4) specifies that exactly one generation is present in the dataspace at a time. Formally, the four properties can be stated as follows:

(SI1) $\quad$ **inv.** $\mathcal{S}(g) \wedge g < N \Leftrightarrow \Xi(\gamma_g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_g :: \neg\Xi(\gamma) \,\rangle$

(SI2) $\quad$ **inv.** $\mathcal{S}(g) \wedge g = N \Leftrightarrow \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} :: \neg\Xi(\gamma) \,\rangle$

(SI3) $\quad$ $\mathcal{S}(g) \wedge g \leq N$ **ensures** $\mathcal{S}(2g)$

(SI4) $\quad$ **inv.** $\langle\, \exists\, g : 1 \leq g \leq N :: \mathcal{S}(g) \,\rangle$

¶ **Proof of (SI1):** **inv.** $\mathcal{S}(g) \wedge g < N \Leftrightarrow \Xi(\gamma_g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_g :: \neg\Xi(\gamma) \,\rangle$

a) First, we show that (SI1) is initially true. This is the case because the initial dataspace contains only the data tuples representing the nodes at generation 1 and the transactions necessary to create the nodes at generation 2 with all these transactions combined into the synchronic group γ_1. Thus:

$$\text{INIT} \Rightarrow \mathcal{S}(1) \wedge \Xi(\gamma_1) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_1 :: \neg\Xi(\gamma) \,\rangle \Rightarrow \text{(SI1)}$$

b) Second, we prove that any synchronic group γ satisfies the property:

$$\{\ \text{(SI1)}\ \}\ \gamma\ \{\ \text{(SI1)}\ \}$$

This property is trivially satisfied by any $\gamma \neq \gamma_g$ because, since such a γ is never enabled, it cannot alter the program state. In the remainder we concern ourselves only with the execution of γ_g and show that the invariant is maintained for all values of $g < N$. Assuming that (SI1) holds before executing γ_g, from the definition of (SI1), the pre-assertion of (SD3) is true. Thus, by applying (IR1) to the property (SD3), we get:

(SD4)
$$\{\ \mathcal{S}(g) \wedge \Xi(\gamma_g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_g :: \neg\Xi(\gamma) \,\rangle\ \}$$
$$\gamma_g$$
$$\{\ \mathcal{S}(2g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_{2g} :: \neg\Xi(\gamma) \,\rangle \wedge (\Xi(\gamma_{2g}) \Leftrightarrow 2g < N)\ \}$$

Based on the last conjunct of the post assertion of (SD4) and given that we assumed $g < N$, we now divide our proof between the cases $2g < N$ and $2g = N$. If $2g < N$, the post assertion corresponds to (SI1) which is thus maintained by executing γ_g. If $2g = N$, we have

$$2g = N \wedge \mathcal{S}(2g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_{2g} :: \neg\Xi(\gamma) \,\rangle \wedge (\Xi(\gamma_{2g}) \Leftrightarrow 2g < N)$$

$\Rightarrow$

$$\neg(\mathcal{S}(2g) \wedge 2g < N) \wedge \neg(\Xi(\gamma_{2g}) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \gamma \neq \gamma_{2g} :: \neg\Xi(\gamma) \,\rangle)$$

$\Rightarrow$

(SI1)

Combining these two cases, we have { (SI1) } γ_g { (SI1) } □

¶ Proof of (SI2): inv. $\mathcal{S}(g) \wedge g = N \Leftrightarrow \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} :: \neg\Xi(\gamma) \,\rangle$

a) The initial dataspace contains the synchronic group γ_1 and its tuples satisfy $\mathcal{S}(1)$, making both sides of the equivalence relation in (SI2) initially false. Thus, INIT $\Rightarrow$ (SI2).

b) Similarly to (SI1), we need to prove { (SI2) } γ { (SI2) } for all γ. Since any group of transactions that does not equal some γ_g is never enabled, we are only concerned with proving that the invariant is maintained by executing γ_g for $g < N$. From the definition and proof of (SI1), the synchronic group γ_{2g} is enabled after executing γ_g as long as $2g < N$, or $g < \frac{N}{2}$, and so will maintain (SI2) since it evaluates to "*false* $\Leftrightarrow$ *false*". From (SD3) if $g = \frac{N}{2}$, executing γ_g results in a state in which $\mathcal{S}(N)$ is true and no synchronic group is enabled, i.e., both sides of the equivalence relation in (SI2) become true. Finally, once $\mathcal{S}(N)$ is established, no synchronic group is enabled and so, from (IR2), any property is maintained. □

¶ Proof of (SI3): $\mathcal{S}(g) \wedge g < N$ **ensures** $\mathcal{S}(2g)$

Based on properties (SI1) and (SI2), (SD4) can be simply written as

(SD5) { $\mathcal{S}(g) \wedge g < N$ } γ_g { $\mathcal{S}(2g)$ }

Moreover, from the definition of $\mathcal{S}$, we have

$$\mathcal{S}(g) \Leftrightarrow \mathcal{S}(g) \wedge \neg\mathcal{S}(2g)$$

while logic gives us

$$\mathcal{S}(2g) \Rightarrow (\mathcal{S}(g) \wedge g < N) \vee \mathcal{S}(2g)$$

Combining these relations with (SD5) yields:

$$\{\, (\mathcal{S}(g) \wedge g < N) \wedge \neg\mathcal{S}(2g) \,\}\ \gamma_g\ \{\, (\mathcal{S}(g) \wedge g < N) \vee \mathcal{S}(2g) \,\}$$

which, from (IR3), proves $\mathcal{S}(g) \wedge g < N$ **unless** $\mathcal{S}(2g)$. Based on (SD1), (SD2), (SI1), and (SI2), we have:

$$\mathcal{S}(g) \wedge g < N \Rightarrow\ E(\mathrm{Sum}(2g,N)) \wedge \mathrm{Sum}(2g,N) \in \gamma_g \wedge \Xi(\gamma_g) \wedge \langle\, \forall\, \gamma : \gamma \subseteq \mathbf{A} \wedge \mathrm{Sum}(2g,N) \in \gamma \wedge \Xi(\gamma) :: \gamma = \gamma_g \,\rangle$$

Combining this property with $\mathcal{S}(g) \wedge g < N$ **unless** $\mathcal{S}(2g)$ and applying (IR4), we obtain (SI3). □

¶ **Proof of (SI4): inv.** $\langle \exists\ g : 1 \le g \le N :: \mathcal{S}(g) \rangle$

To prove this property, we first expand the **unless** part of (SI3):

(SI3a) $\quad \mathcal{S}(g) \wedge g < N$ **unless** $\mathcal{S}(2g)$

with

(SI3b) $\quad \mathcal{S}(g) \wedge g = N$ **unless** False

which is vacuously true, since $\mathcal{S}(N)$ being true implies that no synchronic group is enabled and so any **unless** property is established (from (IR2)). Combining (SI3a) and (SI3b), we get

$\mathcal{S}(g)$ **unless** $\mathcal{S}(2g) \wedge 2g \le N$

$\Rightarrow$ [definition of $\mathcal{S}(g)$]

$\langle \forall\ g : 1 \le g \le N :: \mathcal{S}(g)$ **unless** $(\mathcal{S}(2g) \wedge 2g \le N) \wedge \neg\mathcal{S}(g) \rangle$

$\Rightarrow$ [general disjunction rule for **unless**]

$\langle \exists\ g : 1 \le g \le N :: \mathcal{S}(g) \rangle$ **unless** $\langle \forall\ g : 1 \le g \le N :: \neg\mathcal{S}(g) \rangle \wedge \langle \exists\ g : 1 \le g \le N :: \mathcal{S}(2g) \wedge 2g \le N \rangle$

$\Rightarrow$ [calculus]

$\langle \exists\ g : 1 \le g \le N :: \mathcal{S}(g) \rangle$ **unless** False □

¶ **Proof of (P1): inv.** Sum(T) = SumA

This is initially the case as the only data tuples in the dataspace are the first generation of nodes containing the values of the array. Every execution of a synchronic group γ_g sets to $\perp$ the value of the nodes in the current generation (removed from the dataspace) and a new generation is created. Each new node has a value equal to the sum of its two children. Since every node in a generation is the child of a node in the next generation (except for the root) and no node is the child of two other nodes, the sum of the values of one generation is propagated to the next □

¶ **Proof of (P2.1):** $\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$

To prove (P2.1), we first relate the values of $\mathcal{M}$ and $\mathcal{S}$ as follows:

(SI6) $\quad$ **inv.** $\mathcal{M} = k \Rightarrow \mathcal{S}(g(k))$

where g is defined as: $\quad g(k) = \dfrac{N^2}{N+k}$

Property (P2.1) is proven as follows:

1) $\mathcal{M} = k \wedge k > 0 \Rightarrow \mathcal{S}(g(k)) \wedge g(k) < N$ [from (SI6)]

2) $\mathcal{S}(g) \wedge g < N \mapsto \mathcal{S}(2g)$ [from (SI3)]

3) $\mathcal{S}(2g(k)) \Rightarrow \mathcal{M} = g^{-1}(2g(k)) < k$ [from the definitions of g and $\mathcal{M}$, and calculus] □

¶ Proof of (P3.1): stable $\mathcal{M} = 0$

$\mathcal{M} = 0$ implies that only the root of T has a defined value which, in turn, implies that no group is executable. Thus, from (IR2), the property $\mathcal{M} = 0$ is maintained.

$\mathcal{M} = 0$

$\Rightarrow$ [definition of $\mathcal{M}$]

$val(root(T)) \neq \perp \wedge \langle \forall n : n \in T \wedge gen(n) < N :: val(n) = \perp \rangle$

$\Rightarrow$ [definition of val(n) and γ_k (SI2)]

$\langle \forall g: 1 \leq g \leq N :: \neg\Xi(\gamma_g) \rangle$

$\Rightarrow$ [(IR2)]

$\{ \mathcal{M} = 0 \}\ \gamma\ \{ \mathcal{M} = 0 \}$

$\Rightarrow$ [definition of **stable**]

(P3.1) □

This concludes the correctness proof for the Swarm program. Most of the reasoning made (implicit) use of theorems established for UNITY. The Swarm-specific parts were captured by the properties (SI1-4) that dealt with the relations between the data state and the actions in the Swarm program.

5. Concurrent Processes

5.1. The Model

In the Concurrent Processes (CP) [17] model proposed by Milne and Milner, a computation consists of the evolution of a set of processes. Each process is a set of ports. A port is a triple, written α<v, λz.f>, where α is the name of the port, v its value, and f a continuation function with one argument z. Two ports match if their names are the same except that one has an overbar while the other does not, e.g., α and $\overline{\alpha}$. Since port names are not necessarily disjoint among processes, a port in one process can match ports in several processes. A step of a CP computation is composed of two matching ports exchanging their respective values via a synchronous bidirectional communication, followed by the reconfiguration of the two processes involved in the exchange. Each process is

removed and the result of evaluating the continuation of the communicating port with the value received during the exchange is added to the current set of processes.

In DS each pair of matching ports may be treated as a synchronic group. This is possible because DS allows an action to be part of multiple synchronic groups. By contrast, Swarm's synchrony relation (~) allows only disjoint synchronic groups because synchronic group membership is specified by an equivalence relation, the transitive reflexive closure of the synchrony relation. Because the array summation example makes use of rather simple processes multiple matches are not involved. Nevertheless the DS logic would have no difficulty handling them, if they were present. Actually, we only consider processes composed of a single port. As a result, we do not name the processes but simply represent them as singleton sets, e.g., { α<v, λz.f> }. For this reason, we use the terms process and port interchangeably.

Processes

These are the types of processes that can exist during a computation

{ $Sum_{g,p}$<v, λz.$f_{v,g,p}$> }, { $\overline{Sum}_{g,p}$<v, λz.$\perp$> } *Matching ports for non-root nodes*

{ Result<v, λz.$\perp$> } *The root node*

Definition

For every pair of matching ports, one has a null continuation while the other handles the creation of the parent node. If the parent is the root node, a process with a Result port is created. Otherwise, a Sum or $\overline{Sum}$ *process is created based on the position of the parent node in the tree. In any case, the process whose continuation was evaluated is destroyed.*

$f_{v,g,p}(z)$ = if (g = N) then { Result<v+z, λy.$\perp$> }

else if (p mod (2*g) = 0) then { $Sum_{2*g,p}$<v+z, λy.$f_{v+z,2*g,p}$> }

else { $\overline{Sum}_{2*g,2*p}$<v+z, λy.$\perp$> }

Initialization

Initially, only the processes corresponding to the leaf nodes are created. Each one holds one value of the array and every pair of sibling nodes have matching ports.

$\langle \forall i : 1 \le i \le \frac{N}{2} :: \{ Sum_{2,2*i}<A[2*i], \lambda z.f_{A[2*i],1,2*i}> \}, \{ \overline{Sum}_{2,2*i}<A[2*i-1], \lambda z.\perp> \} \rangle$

Fig. 3. Array summation in CP

Fig. 3 shows a CP definition of the array summation problem. Each node of the computation tree is represented by a process whose port value is the value of the node. Each pair of sibling nodes in the tree has matching ports $Sum_{g,p}$ and $\overline{Sum}_{g,p}$, where g and p are, respectively, the generation and position of the common parent of these nodes. The "barred" port has a null continuation (represented by $\perp$) while the other port's

continuation, denoted by $f_{v,g,p}$ handles the creation of the process representing the parent node with a value equal to the sum of the two sibling values. Finally, the root node is represented by a special process having the port named *Result*. This process could be used to propagate the total sum to some other computation.

Fig. 4 shows the different processes that are created during the computation of the Array Summation for N = 8. Each process is represented by a half-circle with the name of the port. Matching ports are linked by an arc. In the Swarm solution, a step involved the computation of the values of all the nodes in a given generation, resulting in a highly synchronous sweep from the leaf nodes to the root of the tree. By contrast, each step of the CP solution consists only in the evaluation of the value of one node. This makes for a more asynchronous computation. Within a single generation some nodes may have defined values while others may not.

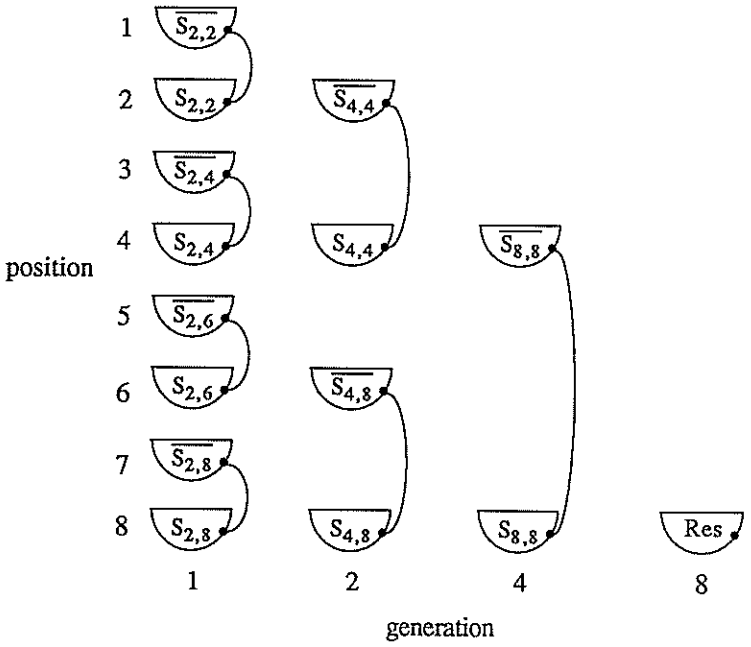

Fig. 4. Example of process instances during an array summation

5.2. Correctness Proof

As was the case for the Swarm program, only nodes with defined values are explicitly represented by some process, as described by the following coupling invariant:

(CD1)

$$\text{val}(n) = v \wedge v \neq \perp \equiv \begin{cases} \text{Result}\langle v, \lambda z.\perp\rangle & \text{if } gen(n) = N \\ \text{Sum}_{2gen(n),pos(n)}\langle v, \lambda z.f_{v,gen(n),pos(n)}\rangle & \text{if } gen(n)<N \wedge (pos(n) \bmod 2gen(n) = 0) \\ \overline{\text{Sum}}_{2gen(n),pos(n)+gen(n)}\langle v, \lambda z.\perp\rangle & \text{if } gen(n)<N \wedge (pos(n) \bmod 2gen(n) \neq 0) \end{cases}$$

where we use the port name to denote the fact that a process having that port is present in the process set. Given the CP execution model, the only feasible groups involve matching ports. In this example, a feasible group corresponds to the actions denoted by the ports $Sum_{g,p}$ and $\overline{Sum}_{g,p}$. For the remainder of the proof, we define such a group as γ_n, where n is the parent of the two nodes represented by the members of the group, and use the ports as action names. More formally:

(CD2) $\quad \gamma_n \equiv \{\ \overline{\text{Sum}}_{gen(n),pos(n)}\langle v, \lambda z.\bot\rangle,\ \text{Sum}_{gen(n),pos(n)}\langle v, \lambda z.f_{v,gen(n),pos(n)}\rangle\ \}$

Based on (CD1), a group γ_n is enabled only if the processes associated with the children of node n are enabled and is feasible only if it corresponds to a pair of matching ports. Thus, we can derive the following two invariants:

(CI1) $\quad \textbf{inv.}\ \langle\ \forall\ \gamma : \gamma \subseteq \mathbf{A} :: \Phi(\gamma) \Rightarrow \langle\ \exists\ n : n \in T \wedge \neg\text{leaf}(n) :: \gamma = \gamma_n\ \rangle\ \rangle$

(CI2) $\quad \textbf{inv.}\ \Xi(\gamma_n) \Leftrightarrow \text{val}(\text{left}(n)) \neq \bot \wedge \text{val}(\text{right}(n)) \neq \bot$

The effect of performing an exchange between two processes is captured by the following property of the synchronic group γ_n (derived from the definition of the continuation function in Fig. 3):

(CD3) $\quad \{\ \text{val}(n) = \bot \wedge \text{val}(\text{left}(n)) = v_l \wedge \text{val}(\text{right}(n)) = v_r \wedge v_l \neq \bot \wedge v_r \neq \bot \wedge \text{def}(T) = \tau\ \}$

$$\gamma_n$$

$$\{\ \text{val}(n)=v_l+v_r \wedge \text{val}(\text{left}(n))=\bot \wedge \text{val}(\text{right}(n))=\bot \wedge (\text{def}(T) = \tau - \{\ \text{left}(n), \text{right}(n)\ \} \cup \{\ n\ \})\}$$

where *def(T)* is the set of defined nodes in the tree *T*, i.e.:

$$\text{def}(T) = \langle\ \textbf{set}\ n : n \in T \wedge \text{val}(n) \neq \bot :: n\ \rangle$$

¶ **Proof of (P1): inv.** Sum(T) = SumA

Initially, the only nodes defined are the leaf nodes whose values correspond to the original array. Thus INIT $\Rightarrow$ Sum(T) = SumA. As described by (CD3), each step of the computation involves two sibling nodes being eliminated while their parent is created with a value equal to the sum of the siblings' values. Given the definition of Sum(T), its value is maintained after each step. □

¶ **Proof of (P2.1):** $\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$

In order to prove (P2.1), we need a property specific to the manner in which progress is accomplished by the CP system:

(CP2.1) $\quad \text{val}(n) = \bot \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0$ **ensures** $\mathcal{M} < k$

Of course, we need to show that the left-hand side of (P2.1) implies the left-hand side of (CP2.1), i.e.:

$$\mathcal{M} = k \wedge k > 0 \Rightarrow \langle\ \exists\ n : n \in T \wedge \neg\text{leaf}(n) :: \Xi(\gamma_n) \wedge \text{val}(n) = \bot\ \rangle$$

This is obtained through the use of another metric, this one counting the number of nodes in the tree whose value is defined, i.e.:

(CD4) $\mathcal{C}(\text{tree}(n), c) \equiv \langle \# m :: m \in \text{tree}(n) \wedge \text{val}(m) \neq \perp \rangle = c$

From the definition of $\mathcal{C}$, we trivially have that a tree for which $\mathcal{M}$ is not zero has some defined nodes, i.e.,

$$\mathcal{M} > 0 \Rightarrow \mathcal{C}(T, c) \wedge c > 1$$

What remains to be proven is the property that a tree in which more that one node is defined contains an undefined node whose children are both defined, i.e.:

(CI3) **inv.** $\mathcal{C}(\text{tree}(n), c) \wedge c > 1$

$$\Rightarrow \langle \exists m : m \in \text{tree}(n) :: \text{val}(m) = \perp \wedge \text{val}(\text{left}(m)) \neq \perp \wedge \text{val}(\text{right}(m)) \neq \perp \rangle$$

which, from the definition of a group γ_n, can also be expressed as

(CI3') **inv.** $\mathcal{C}(\text{tree}(n), c) \wedge c > 1 \Rightarrow \langle \exists m : m \in \text{tree}(n) :: \text{val}(m) = \perp \wedge \Xi(\gamma_m) \rangle$ □

We prove (CI3) by induction using properties (CI4) through (CI6) below which state that 1) a tree whose root is defined has all undefined values for all its other nodes, 2) if only one node in a tree has a value defined, it is the root of the tree and, 3) a tree with more than one defined value is composed of two root subtrees, each with at least one defined value:

(CI4) **inv.** $\langle \forall n : n \in T :: \text{val}(n) \neq \perp \Rightarrow \langle \forall m : m \in \text{tree}(n) \wedge m \neq n :: \text{val}(m) = \perp \rangle \rangle$

(CI5) **inv.** $\langle \forall n : n \in T :: \mathcal{C}(\text{tree}(n), 1) \Leftrightarrow \text{val}(n) \neq \perp \rangle$

(CI6) **inv.** $\mathcal{C}(\text{tree}(n), c) \wedge c>1 \wedge \text{val}(n) = \perp \Leftrightarrow \mathcal{C}(\text{tree}(\text{left}(n)), c_l) \wedge c_l>0 \wedge \mathcal{C}(\text{tree}(\text{right}(n)), c_r) \wedge c_r>0$

(CI5) is a corollary of (CI4).

¶ **Proof of (CI3): inv.** $\mathcal{C}(\text{tree}(n), c) \wedge c > 1$

$$\Rightarrow \langle \exists m : m \in \text{tree}(n) :: \text{val}(m) = \perp \wedge \text{val}(\text{left}(m)) \neq \perp \wedge \text{val}(\text{right}(m)) \neq \perp \rangle$$

For the base case, we use a tree with exactly two defined nodes. From (CI5), the root of this tree is undefined and, based on (CI6), each subtree of the root has at least one defined node. Since the complete tree has only two defined nodes, each root subtree has exactly one defined node and, from (CI5), this node is the root of the subtree. More formally:

$\mathcal{C}(\text{tree}(n), 2)$

$\Rightarrow$ [(CI6) and (CI5)]

$\text{val}(n) = \perp \wedge \mathcal{C}(\text{tree}(\text{left}(n)), 1) \wedge \mathcal{C}(\text{tree}(\text{right}(n)), 1)$

$\Rightarrow$ [(CI5)]

$val(n) = \perp \wedge val(left(n)) \neq \perp \wedge val(right(n)) \neq \perp$

For the induction part, we assume that (CI3) is established for a tree containing less than k defined nodes (with $k > 2$) and consider a tree with k defined nodes. Using the same reasoning as for the base case, we can deduce that the root of the tree is not defined and that both subtrees of the root have at least one defined node and fewer than k defined nodes. Since the tree contains more than two nodes, one of the subtrees contains at least two defined nodes thus satisfying the conditions for (CI3). If one of the subtrees satisfies (CI3), the complete tree also satisfies (CI3).

$\mathcal{C}(tree(n), k) \wedge k > 2$

$\Rightarrow$ [(CI6) and (CI5)]

$\langle \exists\, c_l, c_r :: \mathcal{C}(tree(left(n)), c_l) \wedge \mathcal{C}(tree(right(n)), c_r) \wedge 0 < c_l < k \wedge 0 < c_r < k \wedge (c_l > 1 \vee c_r > 1) \rangle$

$\Rightarrow$ [induction]

$\langle \exists\, m : m \in tree(left(n)) :: val(m) = \perp \wedge val(left(m)) \neq \perp \wedge val(right(m)) \neq \perp \rangle \vee$

$\langle \exists\, m : m \in tree(right(n)) :: val(m) = \perp \wedge val(left(m)) \neq \perp \wedge val(right(m)) \neq \perp \rangle$

$\Rightarrow$ [calculus]

$\langle \exists\, m : m \in tree(n) :: val(m) = \perp \wedge val(left(m)) \neq \perp \wedge val(right(m)) \neq \perp \rangle$ □

¶ **Proof of (CI4): inv.** $\langle \forall\, n : n \in T :: val(n) \neq \perp \Rightarrow \langle \forall\, m : m \in tree(n) \wedge m \neq n :: val(m) = \perp \rangle \rangle$

Initially, the only defined nodes are the leaves of the tree and (CI4) is vacuously satisfied. We now need to prove that any group of actions maintains (CI4). As usual, we are only concerned with executable groups of actions. If γ_p is a synchronic group, we can conclude that all nodes but the children of the root have defined values:

$\Xi(\gamma_p)$

$\Rightarrow$ [from (CI2)]

$val(left(p)) \neq \perp \wedge val(right(p)) \neq \perp$

$\Rightarrow$ [(CI4) assumed true before executing the synchronic group]

$\langle \forall\, m : m \in tree(left(p)) \wedge m \neq left(p) :: val(m) = \perp \rangle \wedge$

$\langle \forall\, m : m \in tree(right(p)) \wedge m \neq right(p) :: val(m) = \perp \rangle$

$\Rightarrow$ [calculus]

$\langle \forall\, m : m \in tree(p) \wedge m \neq p \wedge m \neq left(p) \wedge m \neq right(p) :: val(m) = \perp \rangle$

Moreover, since (CI4) is assumed to hold before executing γ_p, we have $val(p) = \perp$. From (CD3), we see that the effect of executing γ_p is to assign a value to the root of the tree and undefined to the children of the root. Since no other nodes are affected, (CI4) is maintained. □

¶ **Proof of (CI6): inv.** $\mathcal{C}(\text{tree}(n), c) \wedge c>1 \wedge \text{val}(n) = \perp \Leftrightarrow \mathcal{C}(\text{tree}(\text{left}(n)), c_l) \wedge c_l>0 \wedge \mathcal{C}(\text{tree}(\text{right}(n)), c_r) \wedge c_r>0$

This property is initially true since the only defined nodes are the leaves of the tree, and so any subtree with an undefined root (an internal node) has two child subtrees with some defined nodes (the leaves). For any executable group γ_p, we have the following property:

$$\{\ \mathcal{C}(\text{tree}(n), c_n) \wedge \text{val}(n) = v_n \wedge \mathcal{C}(\text{tree}(\text{left}(n)), c_l) \wedge \mathcal{C}(\text{tree}(\text{right}(n)), c_r)\ \}$$

$$\gamma_p$$

$$\{\ \mathcal{C}(\text{tree}(n), c'_n) \wedge \text{val}(n) = v'_n \wedge \mathcal{C}(\text{tree}(\text{left}(n)), c'_l) \wedge \mathcal{C}(\text{tree}(\text{right}(n)), c'_r)\ \}$$

a) If $c_n = 0$ then the subtree rooted at n is entirely composed of undefined nodes and, from (CD2), no synchronic group that could modify it is enabled, thus maintaining (CI6).

b) If $c_n = 1$ then, from (CI4), only the root of the subtree is defined and, from (CD2) and (CD3), the only possible modification to the subtree is the value of the root becoming undefined, maintaining (CI7).

c) If $c_n > 1$ then, from (CI6) holding before executing γ_p, one has:

$$v_n = \perp \wedge c_l > 0 \wedge c_r > 0$$

The effect of executing γ_p depends on four possible locations of p in the tree:

1) $p \notin tree(n)$: If n is not a child of p, then executing γ_p has no effect on $tree(n)$ and based on (CI2), since $\text{val}(n) = \perp$, the synchronic group affecting the value of n is not enabled.

2) $p = n$: From (CI2), $\Xi(\gamma_p) \Rightarrow val(left(n)) \neq \perp \wedge val(right(n)) \neq \perp$. This, based on (CI4), implies that $c_l = 1$ and $c_r = 1$, and, from the definition of $\mathcal{C}$, $c_n = 2$. From (CD3), the effect of executing γ_p is:

$$v'_n \neq \perp \wedge c'_n = 1 \wedge c'_l = 0 \wedge c'_r = 0$$

3) $p \in tree(left(n))$: From (CI2) and the definition of $\mathcal{C}$, $\Xi(\gamma_p) \Rightarrow c_l > 1$. Combined with $c_r > 0$, one has $c_n > 2$. From (CD3), the effect of executing γ_p is:

$$v'_n = \perp \wedge (c'_n = c_n\text{-}1 > 1) \wedge (c'_l = c_l - 1 > 0) \wedge c'_r = c_r$$

4) $p \in tree(right(n))$: Proof similar to case 3). □

¶ **Proof of (CP2.1):** $\text{val}(n) = \perp \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0$ **ensures** $\mathcal{M} < k$

From the rule (IR4), we can prove (CP2.1) if we have the following two properties:

(CP2.1a) $\text{val}(n) = \perp \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0$ **unless** $\mathcal{M} < k$

(CP2.1b) $\langle \exists\, \alpha : \alpha \in A :: \quad (val(n) = \perp \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0 \Rightarrow \langle \exists\, \gamma : \gamma \subseteq A \wedge \alpha \in \gamma :: \Xi(\gamma) \rangle) \wedge$

$\langle \forall\, \gamma : \gamma \subseteq A \wedge \alpha \in \gamma \wedge \Xi(\gamma) ::$

$\{\ val(n) = \perp \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0\ \}\ \gamma\ \{\ \mathcal{M} < k\ \}\ \rangle\ \rangle$

¶ Proof of (CP2.1a): From (CD3), we see that the execution of a synchronic group γ_n results in two nodes becoming undefined and their parent becoming defined. Any executable γ_m (with $m \neq n$) has a similar effect.

¶ Proof of (CP2.1b): Let α be one of the actions in γ_n. The first conjunct of (CP2.1b) is then trivially satisfied for $\gamma = \gamma_n$. Moreover, from (CD2), we know that no node belongs to more than one synchronic group. Thus, the second conjunct of (CP2.1b) is reduced to:

$$\{\ val(n) = \perp \wedge \Xi(\gamma_n) \wedge \mathcal{M} = k \wedge k > 0\ \}\ \gamma_n\ \{\ \mathcal{M} < k\ \}$$

which is directly derived from (CD3) using (IR1). □

¶ Proof of (P3.1): stable $\mathcal{M} = 0$

As in the Swarm program, $\mathcal{M} = 0$ implies that no group is executable, thus the property is trivially maintained. □

This concludes the correctness proof of the CP program. The set of program–specific properties was larger than for the Swarm version mainly because of the more complex computation of the tree of values.

6. Input/Output Automata

6.1. The Model

In the Input/Output Automata (IOA) model [14], a computation is defined by a static (but possibly infinite) set of interacting components. Each component is modelled as an automaton, a mathematical object with a (possibly infinite) set of input, output, and internal operations over a local state typically defined by a set of variables. IOA operations correspond to actions in DS. Output and internal actions are under the control of the automaton and are enabled and disabled by changes in the local state. Output actions are transmitted to the environment (the other components) of the automaton. In contrast, the automaton's input actions originate in the environment and can not be "refused". Several automata can be connected so that the output of one is seen as input by the others. If an output action is executed, the corresponding input actions occur simultaneously. Syntactically, infinite sets of actions are represented by the use of parameterized names such as $\alpha(\nu)$, where the parameter ν ranges over a possibly infinite set and α is a constant.

*The automata are described in terms of their variables and their actions. Each action is composed of a precondition (**pre**) restricting the enabling of the action (implicitly true for an input action) and an effect (eff) on the state of the automaton.*

Automata:

The only purpose of a leaf automaton is to transmit its value (one of the array element) to its parent in the tree. Once the value is set to ⊥, all further output actions are disabled. The action name carries the identity of the originator, its generation and position.

$A_{1,p}$: **var** integer psum := A[p]

output $pass_{1,p}(v)$: **pre:** psum = v ∧ v ≠ ⊥

eff: psum := ⊥

A non-leaf automaton adds the values of its children. It receives the values of the children through two separate parameterized inputs and, after having received both (count = 2), transmits the sum to its parent (or the environment if the node is the root)

$Sum_{g,p}$: **var** integer psum := ⊥, count := 0

input $pass_{\frac{g}{2},p-\frac{g}{2}}(v)$: **eff:** psum, count := psum+v, count+1

$pass_{\frac{g}{2},p}(v)$: **eff:** psum, count := psum+v, count+1

output $pass_{g,p}(v)$: **pre:** psum = v ∧ count = 2

eff: psum, count := ⊥, 0

Structure:

The entire program consists of the automata representing the leaf and internal nodes of the tree. The interconnections are implicit. Input and ouput actions with the same name are executed synchronously.

$\langle \forall p : 1 \le p \le N :: A_{1,p} \rangle \rangle$

$\langle \forall g : 1 < g \le N \wedge \langle \exists k : k > 0 :: g = 2^k \rangle :: \langle \forall p : 1 \le p \le N \wedge p \bmod g = 0 :: Sum_{g,p} \rangle \rangle$

Fig. 5. Array summation in IOA

Fig. 5 shows an IOA solution to the array aummation problem[2] . Each node in the computation tree is represented by an automaton storing the value of the node in its internal state. The leaves initially hold the values of the array. Each transmits its value to its parent by means of an output action which becomes permanently disabled after its execution. The internal nodes receive the values held by their children through two (parameterized) inputs. Each input adds the received value to a local variable (*psum*) and increments a local counter (*count*). The initial *psum* at the internal node is ⊥ and we define the result of adding some value *v* to ⊥ to be equal to *v*. Once the

[2] The notation used here is not exactly that used by Lynch and Turtle in their presentation of I/O Automata. We tried to minimize notational changes from one example to the next.

values of both children have been received, the output action of the automaton is enabled and the partial sum stored at this node can be transmitted to its parent. The CP and IOA programs are quite similar in overall behavior with the only difference being that the single-step summing of sibling nodes' values in CP has been replaced by two separate steps. In addition, it must be noted that the automata are persistent while the CP processes are not. Fig. 6 presents the complete set of automata and the input/output interactions for an array of size N = 8. The output of the root automaton can be connected to some other automaton for further processing of the resulting sum.

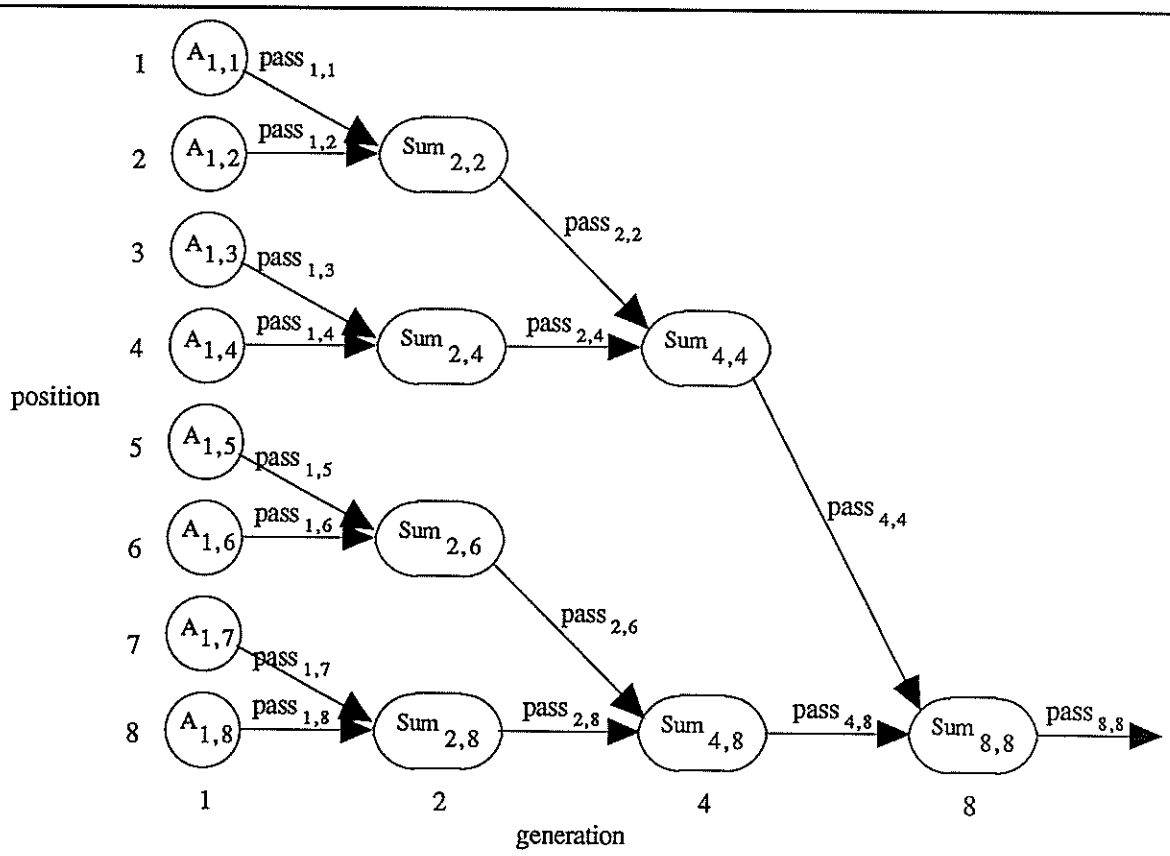

Fig. 6. Example of automata structure for N = 8

6.2. Correctness Proof

In this solution to the Array Summation problem, each node in the computation tree is represented by an automaton which exists throughout the computation. The value of a node is stored in the corresponding automaton's variable named *value*, i.e.:

$$\text{(ID1)} \qquad \text{val(n)} = v \equiv \begin{cases} A_{1,pos(n)}.psum = v & \text{if leaf(n)} \\ Sum_{gen(n),pos(n)}.psum = v & \text{if } \neg\text{leaf(n)} \end{cases}$$

where $\beta.v$ represents variable v of automaton β. In addition, we define a function *count(n)* on an internal node *n* which returns the value of the *count* variable of the corresponding *Sum* automaton. Moreover, for generality and simplicity in the proof, we extend the definition of *count(n)* to the leaf nodes.

$$\text{(ID2)} \qquad \text{count(n)} = \begin{cases} \text{Sum}_{\text{gen(n)},\text{pos(n)}}.\text{count} & \text{if } \neg\text{leaf(n)} \\ 0 & \text{if leaf(n)} \wedge \text{val(n)} = \perp \\ 2 & \text{if leaf(n)} \wedge \text{val(n)} \neq \perp \end{cases}$$

To reason about the behavior of the IOA, we need to view any output action with all its related input actions as a synchronic group. In this example, the output action of a node and the corresponding input action in the parent must form a synchronic group. We use $\gamma_{l(n)}$ and $\gamma_{r(n)}$ to represent, respectively, the synchronic group associated with the values transfer from the left and right children of node *n* to node *n* itself, i.e.:

$$\text{(ID3)} \qquad \gamma_{l(n)} = \begin{cases} \{\ A_{1,p-1}.\text{pass}_{1,p-1}(v),\ \text{Sum}_{2,p}.\text{pass}_{\ 1,p-1}(v)\ \} & \text{if } g = 2 \\ \{\ \text{Sum}_{\frac{g}{2},p}.\text{pass}_{g,p}(v),\ \text{Sum}_{g,p}.\text{pass}_{\ \frac{g}{2},p}(v)\ \} & \text{if } g > 2 \end{cases}$$

$$\text{(ID4)} \qquad \gamma_{r(n)} = \begin{cases} \{\ A_{1,p}.c_{1,p}(v),\ \text{Sum}_{2,p}.c_{\ 1,p}(v)\ \} & \text{if } g = 2 \\ \{\ \text{Sum}_{\frac{g}{2},p}.c_{g,p}(v),\ \text{Sum}_{g,p}.c_{\ \frac{g}{2},p}(v)\ \} & \text{if } g > 2 \end{cases}$$

where $g = gen(n)$ and $p = pos(n)$. Only groups of the form $\gamma_{l(n)}$ or $\gamma_{r(n)}$ are feasible, i.e.:

$$\text{(ID5)} \qquad \textbf{inv.} \quad \langle\ \forall\ \gamma : \gamma \subseteq \mathbf{A} :: \Phi(\gamma) \Leftrightarrow \langle\ \exists\ n : n \in T :: \gamma = \gamma_{l(n)} \vee \gamma = \gamma_{r(n)}\ \rangle\ \rangle$$

While the input actions are always enabled, the output actions are enabled only when values from all children have been received, i.e., when the *count* function returns 2:

$$\text{(ID6)} \qquad \textbf{inv.}\ E(\gamma_{l(n)}) \Leftrightarrow \text{count(left(n))} = 2$$

$$\text{(ID7)} \qquad \textbf{inv.}\ E(\gamma_{r(n)}) \Leftrightarrow \text{count(right(n))} = 2$$

The combined effect of the output and input actions in a group is expressed by properties (ID8) and (ID9). In each case the output action resets the automaton's *psum* value to $\perp$ and (for an internal node) the counter *count* to 0 while the input action adds the value transmitted by the output action to *psum* in the receiving automaton and increments its counter by one. More formally:

$$\text{(ID8)} \qquad \{\ \text{val(n)} = v_n \wedge \text{count(n)} = c_n \wedge \text{val(left(n))} = v_l \wedge \text{count(left(n))} = 2\ \}$$

$$\gamma_{l(n)}$$

$$\{\ \text{val(n)} = v_n + v_l \wedge \text{count(n)} = c_n + 1 \wedge \text{val(left(n))} = \perp \wedge \text{count(left(n))} = 0\ \}$$

(ID9) $\{ \text{val}(n) = v_n \wedge \text{count}(n) = c_n \wedge \text{val}(\text{right}(n)) = v_r \wedge \text{count}(\text{right}(n)) = 2 \}$

$\gamma_{r(n)}$

$\{ \text{val}(n) = v_n + v_r \wedge \text{count}(n) = c_n + 1 \wedge \text{val}(\text{right}(n)) = \perp \wedge \text{count}(\text{right}(n)) = 0 \}$

Based on (ID3), (ID8), and (ID9), the following invariant can be shown to hold:

(II1) **inv.** $\text{val}(n) = \perp \Leftrightarrow \text{count}(n) = 0$

Properties (CI3) and (CI4) for the CP solution have related counterparts (II2) and (II3) for the IOA program. (II2) states that if the sum has not reached the root of T, then some automaton is ready to output its value (i.e., some synchronic group is enabled), while (II3) states that once a node *n* has received two values, all other nodes in the tree rooted at *n* have undefined values:

(II2) **inv.** $\mathcal{M} > 0 \Leftrightarrow \langle \exists\, n : n \in T :: \text{count}(\text{left}(n)) = 2 \vee \text{count}(\text{right}(n)) = 2 \rangle$

(II3) **inv.** $\text{count}(n) = 2 \Rightarrow \langle \forall\, m : m \in \text{tree}(n) \wedge m \neq n :: \text{val}(m) = \perp \rangle$

The proofs of these two properties are very similar to their (CI3) and (CI4) counterparts and are omitted for the sake of brevity.

¶ **Proof of (P1): inv.** Sum(T) = SumA

Initially, only the leaf nodes have defined values and hold the entire array A. Whenever $\gamma_{l(n)}$ or $\gamma_{r(n)}$ is executed, the value stored at the child is transfered to the parent node and is added to the parent's value. At the same time, the child resets its value to $\perp$. This execution does not change the value of Sum(T). □

¶ **Proof of (P2.1):** $\mathcal{M} = k \wedge k > 0 \mapsto \mathcal{M} < k$

This is proved by applying the disjunction rule for $\mapsto$ to property (IP2.1):

(IP2.1) $\langle \forall\, n : n \in T \wedge \neg\text{leaf}(n) :: \mathcal{M} = k \wedge k > 0 \wedge (\text{count}(\text{left}(n)) = 2 \vee \text{count}(\text{right}(n)) = 2) \mapsto \mathcal{M} < k \rangle$

and reducing the left-hand side of the resulting $\mapsto$ expression to $\mathcal{M} = k \wedge k > 0$ based on (II2). □

¶ **Proof of (IP2.1):** $\mathcal{M} = k \wedge k > 0 \wedge (\text{count}(\text{left}(n)) = 2 \vee \text{count}(\text{right}(n)) = 2) \mapsto \mathcal{M} < k$

(IP2.1) is the disjunction of the following two properties:

(IP2.1a) $\mathcal{M} = k \wedge k > 0 \wedge \langle \exists\, n : n \in T :: \text{count}(\text{left}(n)) = 2 \rangle \mapsto \mathcal{M} < k$

(IP2.1b) $\mathcal{M} = k \wedge k > 0 \wedge \langle \exists\, n : n \in T :: \text{count}(\text{right}(n)) = 2 \rangle \mapsto \mathcal{M} < k$ □

¶ **Proof of (IP2.1a):** $\mathcal{M} = k \wedge k > 0 \wedge \langle \exists\, n : n \in T :: \text{count}(\text{left}(n)) = 2 \rangle \mapsto \mathcal{M} < k$

(IP2.1a) is directly derived from:

(IP2.1a') $\mathcal{M} = k \wedge k > 0 \wedge \langle \exists n : n \in T :: count(left(n)) = 2 \rangle$ **ensures** $\mathcal{M} < k$

As mentioned before, executing a synchronic group results in the child node resettting its value to $\perp$ while the parent's value is equal to or is augmented by the value received from the child. In any case, this decreases the value of $\mathcal{M}$. Only synchronic groups which reduce $\mathcal{M}$ can be executed. Thus, we have:

$$\mathcal{M} = k \wedge k > 0 \wedge \langle \exists n : n \in T :: count(left(n)) = 2 \rangle \textbf{ unless } \mathcal{M} < k$$

To prove (IP2.1a'), we use (IR4) with α replaced by the "child" node of synchronic group $\gamma_{l(n)}$. From (ID1), (ID2), and (ID3), we trivially have

$$\mathcal{M} = k \wedge k > 0 \wedge \langle \exists n : n \in T :: count(left(n)) = 2 \rangle \wedge \neg(\mathcal{M} < k) \Rightarrow \Xi(\gamma_{l(n)}) \wedge \alpha \in \gamma_{l(n)}$$

Moreover, since the only enabled synchronic group containing α satisfies the property

$$\{ (\mathcal{M} = k \wedge k > 0 \wedge \langle \exists n : n \in T :: count(left(n)) = 2 \rangle) \wedge \neg(\mathcal{M} < k) \} \ \gamma_{l(n)} \ \{ \mathcal{M} < k \}$$

we have proven (IP2.1a'). □

¶ **Proof of (IP2.1b):** $\mathcal{M} = k \wedge k > 0 \wedge \langle \exists n : n \in T :: count(right(n)) = 2 \rangle \mapsto \mathcal{M} < k$

This proof is identical to the one for (IP2.1a) with *right*(n) and $\gamma_{r(n)}$ substituted for *left*(n) and $\gamma_{l(n)}$, respectively, in the proof. □

¶ **Proof of (P3.1): stable** $\mathcal{M} = 0$

As in the Swarm program, $\mathcal{M} = 0$ implies that no group is executable, thus the property is trivially maintained. □

This completes the third proof of a program solving the Array Summation problem. Parts of earlier proofs were reused due to the similarity in the structure and the behavior of the CP and IOA programs. As before, once we formalized the use of synchrony in this model, the DS logic was easily applied to show that the IOA program satisfies the requirements of the problem.

7. Conclusions

This research was motivated by our earlier work on Swarm. The ability to alter dynamically the structure of synchronous computations provides an interesting modeling capability of potential import in specifying and reasoning about systems that combine synchronous and asynchronous features—particularly when the system may be subjected to dynamic reconfiguration. However the Swarm mechanism for specifying synchronic groups, the synchrony relation, proved to be less general that we expected. For instance, the requirement that an action belong to a single synchronic group at a time made it difficult to simulate synchronous communication between matching

ports when one port may have several possible matches. Nevertheless, the successful development of the Swarm proof logic showed us how to deal with dynamic forms of synchrony and laid the foundation for seeking a more general and abstract treatment.

We believe that the *Dynamic Synchrony* model is both general and abstract: 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. We attempted to demonstrate the generality of the model by showing how Dynamic Synchrony can be used to verify three versions of a parallel array summation problem expressed in three different models of concurrency that offer very distinct treatments of synchrony. Swarm is a shared dataspace model in which synchronic groups are specified explicitly and form a partition over the set of enabled actions. Concurrent Processes, the predecessor of CCS[18], allows for synchronous data exchanges between actions associated with pairs of matching ports. Input/Output Automata requires the output action of one automaton to be synchronized with all the input actions bearing the same name in all the other automata. The fact that the verification did not require any sort of translation of the three programs into a new single notation is testimony to the abstract nature of our model.

References:

[1] Andrews, G. R., and Reitman, R. P., "An Axiomatic Approach to Information Flow in Programs," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 56,76, 1980.

[2] Apt, K. R., Francez, N., and Roever, W. P. D., "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 3, pp. 359-385, 1980.

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

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

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

[6] Floyd, R. W., Eds., *Assigning meanings to programs* , 19, American Mathematical Society, Providence, R.I., 1967.

[7] Gamble, R. F., Roman, G.-C., and Ball, W. E., "Formal Verification of Pure Production System Programs," *Proceedings of the 9th National Conference on Artificial Intelligence*, vol. 1, pp. 329-334, 1991.

[8] Gries, D., *The Science of Programming*, Springer-Verlag, New York, NY, 1981.

[9] Hillis, W. D., and Guy L. Steele, J., "Data Parallel Algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170-1183, 1986.

[10] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, pp. 576-580,583, 1969.

[11] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, 1978.

[12] Lamport, L., "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125-143, 1977.

[13] Lamport, L., "The "Hoare Logic" of Concurrent Programming," *Acta Informatica*, no. 14, pp. 21-37, 1980.

[14] Lynch, N. A., and Tuttle, M. R., "An Introduction to Input/Output Automata," *CWI Quaterly*, vol. 2, no. 3, pp. 219-246, 1989.

[15] Manna, Z., and Pneuli, A., "Verification of Concurrent Programs: the Temporal Framework," in *The Correctness Problem in Computer Science*, R. S. Boyer, J. S. Moore, Eds., Academic Press, London, pp. 141-154, 1981.

[16] Manna, Z., and Pnueli, A., "Axiomatic Approach to Total Correctness of Programs," *Acta Informatica*, vol. 3, no. 243-264, 1974.

[17] Milne, G., and Milner, R., "Concurrent Processes and Their Syntax," *Journal of the Association for Computing Machinery*, vol. 26, no. 2, pp. 302-321, 1979.

[18] Milner, R., *A Calculus for Communicating Systems*, G. Goos, J. Hartmanis, Eds., Lecture Notes in Computer Science, Springer-Verlag, New York, NY, vol. 92, 1980.

[19] Owicki, S., and Gries, D., "Verifying properties of parallel programs: an axiomatic approach," *Communications of the ACM*, vol. 19, no. 5, pp. 279-85, 1976.

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

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