Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-90-36

1991-09-01

Dynamic Synchrony among Atomic Actions

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

Synchrony continues to be an important concern in concurrent programming. Existing languages and models have introduced a great variety of constructs for expressing and managing synchronization among sequential processes or atomic actions. This paper puts forth a model in which synchrony is viewed as a relation among atomic actions, a relation which may evolve with time. The model is shown to be convenient for expressing formally the semantics of synchrony as it appears in many of the languages and models proposed to date. Among such models Swarm is singled out for its use of dynamic synchrony. The Swarm notation... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Roman, Gruia-Catalin; Plun, Jerome Y.; and Wilcox, C. Donald, "Dynamic Synchrony among Atomic Actions" Report Number: WUCS-90-36 (1991). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/709

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/709

Dynamic Synchrony among Atomic Actions

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

Complete Abstract:

Synchrony continues to be an important concern in concurrent programming. Existing languages and models have introduced a great variety of constructs for expressing and managing synchronization among sequential processes or atomic actions. This paper puts forth a model in which synchrony is viewed as a relation among atomic actions, a relation which may evolve with time. The model is shown to be convenient for expressing formally the semantics of synchrony as it appears in many of the languages and models proposed to date. Among such models Swarm is singled out for its use of dynamic synchrony. The Swarm notation is briefly reviewed. A new concurrent algorithm for the leader election problem provides a vehicle for illustrating the use of dynamic synchrony in Swarm.

Dynamic Synchrony among Atomic Actions

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

WUCS-90-36

September 1991

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

Abstract

Synchrony continues to be an important concern in concurrent programming. Existing languages and models have introduced a great variety of constructs for expressing and managing synchronization among sequential processes or atomic actions. This paper puts forth a model in which synchrony is viewed as a relation among atomic actions, a relation which may evolve with time. The model is shown to be convenient for expressing formally the semantics of synchrony as it appears in many of the languages and models proposed to date. Among such models Swarm is singled out for its use of dynamic synchrony. The Swarm notation is briefly reviewed. A new concurrent algorithm for the leader election problem provides a vehicle for illustrating the use of dynamic synchrony in Swarm.

Keywords: dynamic synchrony, concurrent languages, concurrent models, concurrent algorithms.

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

(314) 935-6190
roman@CS.WUSTL.edu
fax: (314) 935-7302

1. Introduction

Synchrony has been a primary concern of the distributed computing community throughout its history. Synchronous modes of computing and communication have been studied intensely. Synchronous communication is central to several prominent models of concurrency and emerging languages, including Ada [1]. Synchronous computation is quintessential in modern SIMD architectures, in systolic arrays, and in most VLSI designs. Many classical distributed computing problems have solutions which assume a synchronous mode of computation, e.g., leader election [4]. Distributed system design methodologies that use synchronous centralized solutions as the starting point for system development have recently gained considerable attention. The broad interest in synchrony is, in part, a natural consequence of the increased degree of predictability which synchronous computation and communication offer over the asynchronous alternatives. In the synchronous case, highly abstract specifications are more readily constructed and algorithms are simpler to specify, analyze, and verify. As one might expect, the hidden disadvantage is an increase in the implementation complexity of programming languages that make use of synchronous constructs—asynchronous computing and communication are more easily realized.

In the programming language arena, *CSP* [10] has been instrumental in promoting a general model of distributed systems in which computation is carried out by a static set of sequential processes and communication (including pure synchronization) is accomplished via blocking, asymmetric, synchronous, two-party interactions called *Input/Output Commands*. For the most part, subsequent work has attempted to generalize the input/output commands. In both the *Shared Actions* [12] and the *Multiway Rendezvous* [5] models, communication is captured by a sequential program that modifies the combined state of a fixed set of participating processes. In the *Multiparty Interactions* model [7], only processes that participate in a communication are permitted to read the combined state, all modifications being local and taking place at the end of the interactions. Despite these variations, all these constructs exhibit blocking, are restricted to statically-defined participation in each communication, and are based upon the process concept.

Newly emerging models of concurrency see the process concept as a vestige of sequential programming. The *Input/Output Automata* model [11] replaces the sequential process by a nondeterministic infinite-state machine;

each output command has a single source and all recipients of that command are always ready to accept it—thus the only blocking is by input commands waiting for a matching output command. The *Action Systems* model [2] sees a process as a locus associated with a set of variables; an action involving a single process defines a computation, while one involving multiple processes represents a communication. Although actions can not execute if disabled, there is no blocking in the sense of waiting for some other party to be ready. The *UNITY* model [3] and the *Swarm* model [13] go one step further; an assignment statement or transaction, respectively, is executed merely because it exists. The underlying semantics are those of a state-transition system. The distinction between communication and computation is a matter of interpretation. The sequential process concept and constructs have been eliminated altogether.

Is this the end of synchronous communication? Not at all—UNITY and Swarm make extensive use of synchrony by allowing for synchronous execution of assignment statements and transactions. Nevertheless, new concepts and definitions are needed to encompass both process-centered and action-centered models of distributed computation. One such definition is put forth in this paper. *Synchrony* is defined as the coordinated execution of a set of atomic actions. The participating actions are specified by an equivalence relation called the *synchrony relation*. A statically-defined synchrony relation (*static synchrony*) is adequate for capturing the forms of synchrony in most models of distributed computation. Dynamic redefinition of the synchrony relation *(dynamic synchrony)* is required to accommodate Swarm, the model that provided the initial motivation for this work.

The main body of the paper is organized as follows. Section 2 formalizes the concept of dynamic synchrony. Section 3 demonstrates the appropriateness of the definition via a series of examples which relate it to forms of synchrony already familiar to many readers. Section 4 introduces the Swarm notation—the only model to date that makes use of dynamic synchrony and has an associated assertional-style proof logic. Section 5 presents a new solution—one that exploits dynamic synchrony—to a classical problem, leader election. Section 6 considers technical and practical implications of the dynamic synchrony concept.

2. Semantic Model

In this section we seek to define a semantic model which is general enough to express the types of synchrony present in many concurrent programming languages and models. Our approach is to augment a basic state-transition system by specifying 1) which actions are to be executed synchronously and 2) the effect of such a coordinated execution. Section 2.1 is concerned with the former issue while Sections 2.2 and 2.3 address the latter.

2.1. Definition of synchrony

Let us 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), we characterize the data state of P by the set of data objects currently in existence and the control state 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.

Synchrony is defined as the coordinated execution of one 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 executions are said to be *in synchrony* with each other. If two actions are in synchrony with a third, they are clearly in synchrony with each other. For these reasons, we introduce an equivalence relation, called the *synchrony relation*, to specify which actions are in synchrony with each other at any given time; we make this relation a component of the program state. A convenient abstract representation of the synchrony relation is the partition it induces over the set A, i.e., as a set of disjoint subsets of A whose union equals A. Throughout the paper we will ignore the distinction between the synchrony relation and the partition it induces on A and will treat them as if they were the same thing. This results in the following definition of the state space PS of P:

$$PS = \wp_f(D) \times \wp_f(A) \times \Pi(A)$$

where $\wp_f(X)$ —a set of sets—denotes the set of all finite subsets of X and $\Pi(X)$—a set of sets of sets—denotes a set consisting of all possible partitions of X.

We use σ^d, σ^a, and σ^s to denote the three components of the program state σ, i.e., the data state, the control state, and the partition induced by the synchrony relation associated with P. Operationally, the transition system executes as follows. In each step, some enabled operation in σ^a is selected for execution together with all the other operations in σ^a which, according to the synchrony relation, belong to the same equivalence class of σ^s. The selected set of enabled operations is called a *synchronic group*. In each state, the synchrony relation partitions the enabled actions into disjoint synchronic groups, where the actions of a particular synchronic group must execute synchronously. We use σ^g to refer to the set of synchronic groups existing in some state σ

$$\sigma^g \equiv \{ \sigma^a \cap \gamma \mid \gamma \in \sigma^s \}$$

In this model there is no blocking; by selecting a particular action for execution all the enabled actions that are part of the same equivalence class are executed. The disabled actions are simply ignored. Also, the selection of the actions to execute next is based on selecting individual actions and not groups which, as shown later, may have a very volatile nature. Finally, we assume that the action selection is weakly fair—in an infinite execution an action that is enabled infinitely often is selected infinitely often.

An execution of P is defined as an alternating sequence of states and synchronic groups

$$\sigma_0 \pi_0 \sigma_1 \pi_1 \sigma_2 \pi_2 \sigma_3 \pi_3 \ \cdots$$

where σ_0 is a valid initial state of the program P and σ_{i+1} is a state which may be reached from σ_i by executing the synchronic group πi. Of course, πi must be an element of σ_i^g, and cannot be empty unless there are no enabled actions. All executions are considered infinite by extending any finite execution with pairs consisting of the final state and the empty set.

Illustration. Consider a program consisting of four sequential processes p_1, p_2, p_3, and p_4 and let a_1, a_2, a_3, and a_4 denote the sets of actions associated with each process. If we require p_1 to synchronize with p_2, and p_3 with p_4 respectively, then the synchrony relation of the entire program is captured by the partition

$$\sigma^s \equiv \{ a_1 \cup a_2 , a_3 \cup a_4 \}.$$

Since each sequential process has a single action enabled at a time, σ^g consists of two pairs of actions: the currently enabled actions of p_1 and p_2 and the currently enabled actions of p_3 and p_4. If the enabled action of p_1 is selected, the enabled action of p_2 is selected by implication and the two actions are executed synchronously. A fully synchronous version of the same program simply requires redefining the synchrony relation such that

$$\sigma^s \equiv \{ a_1 \cup a_2 \cup a_3 \cup a_4 \}.$$

Now all four processes are synchronized on each step; this corresponds to a synchronous execution of the four processes. Whether or not they also communicate with each other depends upon the actions they execute and upon how we define coordinated execution.

2.2. Coordinated execution

We now define the concept of coordinated execution, i.e., the semantics of executing a synchronic group in some particular state. We choose to characterize the execution of a synchronic group as an atomic transformation of the program state (by analogy with the execution of atomic actions). Furthermore, we postulate the existence of a semantic function E which maps each synchronic group into a set of state pairs. Given a group π and a state σ, the pair (σ,σ') is in the set $E[\pi]$ if 1) σ^g contains π, and 2) there are actions to perform. Then, σ' is one of the states that could result from the execution of π in σ

$$\langle \forall\ \sigma,\sigma',\pi : (\sigma, \sigma') \in \mathbb{E}[\pi] :: \pi \in \sigma^g \wedge \sigma^a \neq \emptyset \rangle$$

The empty synchronic group has meaning only when there are no enabled actions

$$\langle \forall\ \sigma : (\sigma, \sigma) \in \mathbb{E}[\ \emptyset\ ] :: \sigma^a = \emptyset \rangle$$

This definition has several advantages. First, a single function captures the effect of executing either individual actions or multiple actions. Second, the definition is general enough to accommodate nondeterminism. Third, the semantic function may be easily customized to the specifics of a particular model. Below we use CSP [10] to illustrate this last point. Additional examples are provided in Section 3.

Illustration. Consider a generic CSP program consisting of *n* processes. The processes execute *local actions* asynchronously; they also execute *communications* requiring pairwise synchronization of two processes. The global data state of the program is fully determined by the current assignment of values to all the local variables. The global control state is fully determined by the actions enabled in each process, i.e., assignments to local variables and/or pending input/output. A local action can alter the local data and control state of the respective process only. A communication action alters the state of the two processes involved. A process may be willing to communicate with one or more other processes, and as a consequence of engaging successfully in any one communication may freely alter the set of communications in which it is willing to take part.

Let a and $\bar{a}$ a represent an arbitrary pair of matching input/output commands. Since the communication is synchronous, it is natural to require that the two commands be executed in synchrony with each other, i.e.,

$$a \approx \bar{a}$$

Here we introduce the notation "≈" to denote the synchrony relation. We treat each pending action as enabled, even though in CSP an unmatched input/output command is blocked. Consequently, the two actions above may appear in three distinct synchronic groups:

$$\{ a \}, \{ \bar{a} \}, \text{and } \{ a, \bar{a} \}.$$

The first two synchronic groups correspond to blocked actions, while the third represents a possible communication.

If the synchronic group $\{a, \bar{a}\}$ is executed, the two processes involved change both data and control state (as easily captured by some appropriate definition of $E[\{a,\bar{a}\}]$). Blocking is modeled by an identity transformation of the program state:

$$(\sigma, \sigma') \in E[\{\pi\}] \wedge (\pi = \{a\} \vee \pi = \{\bar{a}\}) \Rightarrow (\sigma = \sigma')$$

One possible shortcoming of this approach to modeling CSP is the difficulty that arises if, for instance, a process is willing to participate in the same state in two input actions (say a and a') which both match the same output action (say $\bar{a}$) in some other process. Because of the transitivity of the synchrony relation we have

$$a \approx \bar{a} \wedge a' \approx \bar{a} \Rightarrow a \approx a'$$

and the undesirable synchronic groups

$$\{ a, a' \}, \text{and } \{ a, a', \bar{a} \}.$$

We can avoid the problem by splitting the output command into two distinct ones, say $\bar{a}$ and $\bar{a}'$, having identical effects. Another possible approach to solving the problem would be to introduce multiple synchrony relations; however, we do not see yet a strong compelling reason to do it at this time.

2.3. Dynamic synchrony

All the examples presented so far shared the implicit assumption that the synchrony relation is unchanged by the execution of the program. Our model, however, allows actions to modify the synchrony relation, which is viewed as just another component of the program state. In other words, the kind of synchrony supported by our model is *dynamic* in nature. We now examine the problem of specifying the modification of the synchrony relation. Previously, we found it convenient to treat the synchrony relation as a partition over A (the set σ^S), or as an equivalence relation over A (denoted by $\approx$). When the universe of actions is unbounded, neither view can be easily manipulated by a program.

Another approach is to introduce a new relation $\sim$ (called the *coupling relation*), whose reflexive transitive closure is by definition $\approx$. For the sake of minimality we require the coupling relation $\sim$ to be irreflexive and symmetric. Individual actions can alter the synchrony relation indirectly by adding and removing entries in the coupling relation. Notationally, $(\alpha \sim \beta)$ denotes the predicate "*α is coupled to β*".

Using these conventions we can now redefine the state space of P to be

$$PS = \wp_f(D) \times \wp_f(A) \times \wp_f(A \times A)$$

where the synchrony relation is specified indirectly in terms of $\sim$. This has the advantage that the effect of each program action can be treated uniformly as deletions and additions to each component of the program state:

$$(\sigma, \sigma') \in \mathbf{E}[\pi] \Rightarrow \pi \in \sigma^g \wedge \langle \exists \sigma_{del}, \sigma_{add} : \sigma_{del}, \sigma_{add} \in PS :: \sigma' = (\sigma - \sigma_{del}) + \sigma_{add} \rangle$$

where – (+) represents component-wise set difference (union). Because it is often advantageous to refer to deletions and additions directly, an alternate semantic function ***F*** may be introduced as a replacement for ***E***:

$$(\sigma, \sigma') \in \mathbf{E}[\pi] \Leftrightarrow \langle \exists \sigma_{del}, \sigma_{add} : (\sigma, \sigma_{del}, \sigma_{add}) \in \mathbf{F}[\pi] :: \pi \in \sigma^g \wedge \sigma' = (\sigma - \sigma_{del}) + \sigma_{add} \rangle$$

Illustration. Consider the case of a message server *S* which every so often broadcasts a signal to three clients p_1, p_2, and p_3. All processes, the server and the clients, execute asynchronously except for the case of the broadcast which involves some action *srv* in the server and the actions t_1, t_2, and t_3 in the clients. Let us also assume that initially the clients have no interest in listening to the broadcasts but throughout their computation they may choose to participate in or disassociate from the broadcast at will. The initial state of the synchrony relation is the identity relation and the coupling relation is empty.

A client, say p_1, may start listening to the broadcast by coupling its action t_1 with the server action *srv*. Thus, the coupling relation is redefined to contain $(srv \sim t_1)$ and σ^s becomes

{ {srv, t1}, {t2}, {t3}, ... all other actions }

When all clients listen to the broadcasts the coupling relation consists of $(srv \sim t_1)$, $(srv \sim t_2)$, and $(srv \sim t_3)$ while σ^s becomes

{ {srv,t1,t2,t3}, ... all other actions }

Because of the dynamic synchrony, the definition for blocking is somewhat more complex

$$(\sigma, \sigma') \in \mathbf{E}[\{\pi\}] \wedge (srv \in \pi) \wedge \langle \exists \tau : \tau \in \{t1,t2,t3\} :: \tau \approx srv \wedge \tau \notin \pi \rangle \Rightarrow (\sigma = \sigma')$$

3. Expressive Power

In this section we provide several instantiations of our model to illustrate its generality and expressive power. The first example deals with lock-step execution in message passing algorithms. The other three relate our

model of synchrony to multiparty interactions, UNITY, and Swarm. The presentation focusses on ways to express the various forms of synchrony present in these models in terms of the synchrony relation. There is no attempt to develop complete formalizations; formal statements are included only to show that a clean and complete formalization is feasible.

Lock Step Execution. For the purpose of analyzing certain classes of message-passing algorithms it is often convenient to assume that a computation proceeds in rounds with each of the n processes getting one turn per round to act upon all the messages sent to it in the previous round, e.g. [8]. Each process p_i may execute either a communication action α_i or some local action. The effect of α_i is to remove all messages destined for p_i from the data state, and to add new messages to the data state. Because all the communication actions must be synchronized, the synchrony relation is constant and satisfies the property

$$\alpha_i \approx \alpha_j \equiv (1 \leq i \leq n \ \wedge \ 1 \leq j \leq n)$$

Local actions are not in synchrony with any other actions. The semantic function $\boldsymbol{E}$ again must capture the blocking effect that occurs whenever one of the processes is not ready for communication

$$(\sigma, \sigma') \in \mathbf{E}[\{\pi\}] \wedge \pi \in \sigma^g \wedge \langle \exists\, i, j : 1 \leq i \leq n \wedge 1 \leq j \leq n :: \alpha_i \in \pi \wedge \alpha_j \notin \pi \rangle \Rightarrow (\sigma = \sigma')$$

The effect of synchronous access to the message buffers by the entire process society is conveniently given in terms of the following constraint on $\boldsymbol{F}$

$$(\sigma, \sigma_{del}, \sigma_{add}) \in \mathbf{F}[\{\alpha_1, \ldots, \alpha_n\}] \wedge (\{\alpha_1, \ldots, \alpha_n\} \in \sigma^g) \Rightarrow (\sigma_{del} = \sigma.\mathrm{msg})$$

where $\sigma.msg$ denotes the contents of the message buffers in the particular state.

Multiparty Interactions. In its simplest form, this model [7] assumes a set of sequential processes with local variables and a communication mechanism called an *interaction*. Each interaction has a fixed set of participating processes and requires that all processes in the set be ready to execute for the interaction to take place, thus providing blocking synchronization. Each process can be a member of several interactions but can only be involved in one interaction at any one time. The values of the local variables of each participating process are frozen

at the beginning of the interaction and are freely accessed by all participants during the interaction. Each process updates its local variables at the end of the interaction. As in the previous example, we can accommodate this model by synchronizing only those actions that participate in a common interaction

$$\alpha \approx \beta \equiv (\alpha \in \mathrm{I} \wedge \beta \in \mathrm{I} \wedge (\mathrm{I} \text{ is an interaction}))$$

and by blocking if not all participants in the interaction are ready

$$(\sigma,\sigma') \in \mathbf{E}[\![\{\pi\}]\!] \wedge \pi \in \sigma^g \wedge \langle \exists\, \alpha,\beta, \mathrm{I} : \alpha \in \mathrm{I} \wedge \beta \in \mathrm{I} \wedge (\mathrm{I} \text{ is an interaction}) :: \alpha \in \pi \wedge \beta \notin \pi \rangle \Rightarrow \sigma = \sigma'$$

Finally, since each process updates only its local variables (expressed as the removal and insertion of name-value pairs for each affected variable) the effect of an interaction is the sum of the effects of the participating actions

$$(\sigma, \sigma_{del}, \sigma_{add}) \in \mathbf{F}[\![\{\alpha_1, ..., \alpha_n\}]\!] \wedge (\{\alpha_1, ..., \alpha_n\} \in \sigma^g \wedge \{\alpha_1, ..., \alpha_n\} \text{ is an interaction }) \Rightarrow$$

$$(\sigma_{del} = \langle \cup\, i : 1 \le i \le n :: \text{deletions caused by } \alpha_i \rangle \wedge \sigma_{add} = \langle \cup\, i : 1 \le i \le n :: \text{additions caused by } \alpha_i \rangle)$$

Synchrony in UNITY. A UNITY [3] program consists of a fixed set of variables and a fixed set of conditional multiple-assignment statements. The execution is asynchronous with statements being selected for execution in arbitrary order but fairly. Statements may be combined into synchronic groups using the || operator; a multiple-assignment statement can also be represented as a list of single-assignment statements separated by ||. If we model each single-assignment statement by a distinct action, the static synchrony present in a UNITY program assumes the form

$$\alpha \approx \beta \equiv \text{"}\alpha \parallel \beta\text{" appears in the UNITY program explicitly or implicitly in a multiple-assignment}$$

Further, the UNITY control state is also static and contains an action for every single-assignment statement in the program. In other words, all the synchronic groups present in the initial state continue to exist throughout the execution of the program. Finally, we must note that the weak fairness assumption of UNITY is preserved by this representation as long as we do not consider UNITY programs with an unbounded number of single-assignment statements.

Dynamic synchrony. Swarm [13] is a model of concurrent computation which reduces both computation and communication to atomic transformations of a set of tuple-like entities called the dataspace. The dataspace contains data-tuples, transactions, and coupling relation entries. Transactions are executed atomically and may alter any aspect of the dataspace, except that a transaction cannot delete other transactions. A transaction is deleted automatically whenever it is executed unless it reinserts itself explicitly. It is important to note that transactions can alter not only the data and control state of the program but also its synchrony. Swarm is the first model to include this capability and also to provide a proof logic [6, 14] that allows one to reason formally about dynamic synchrony.

Each Swarm transaction may be represented by a finite set of independent actions which are executed synchronously and are inserted into and deleted from the control state as a group. This aspect of the synchrony relation is static. The actions associated with a particular transaction may be dynamically coupled and decoupled with the actions associated with other transactions. This is accomplished by direct manipulation of the coupling relation; each action can test and modify the coupling relation residing in the dataspace.

Another novel feature introduced by Swarm is the ability of one action participating in a synchronous execution to alter its outcome based on the results of the other participating actions. Each action of a synchronic group broadcasts to the entire group a (possibly empty) set of boolean values. Each action in the group examines the union of all broadcast sets to determine whether to contribute any state changes (dataspace deletions and insertions) to the overall effect of the group. This decision is encapsulated by a boolean function, called a *consensus function*, associated with each action and whose argument is a set of boolean values. Semantically, each action in Swarm may be viewed as a function which given a state σ returns a set of four-tuples each consisting of intended deletions from the dataspace, intended insertions into the dataspace, a set of boolean values to be broadcasted to the group, and a consensus function:

$$(\sigma_{del}, \sigma_{add}, \beta, \Theta) \in C[\alpha]\sigma \Rightarrow \sigma_{del} \in PS \wedge \sigma_{add} \in PS \wedge \beta \subseteq \{0,1\} \wedge (\Theta \in \wp(\{0,1\}) \rightarrow \{0,1\})$$

where C is a new semantic function required to express the meaning of each action α.

Since any action could, in principle, be executed synchronously with any other action, Swarm adopts a very uniform definition of coordinated execution, free of any syntactic restrictions—by contrast, models such as UNITY restrict assignment statements that execute synchronously from assigning values to the same variable. The effect of executing a synchronic group is to perform all the deletions associated with the participating actions before performing any insertions; the deletions and insertions associated with actions whose consensus function evaluates to false are ignored:

$$(\sigma,\sigma') \in \mathbb{E}[\{\alpha_1, \alpha_2, \ldots, \alpha_n\}] \Leftrightarrow$$
$$(\exists\, B, \sigma^1_{del}, \sigma^1_{add}, \beta^1, \Theta^1, \ldots, \sigma^n_{del}, \sigma^n_{add}, \beta^n, \Theta^n :$$
$$\langle \forall\, i : 1 \le i \le n :: (\sigma^i_{del}, \sigma^i_{add}, \beta^i, \Theta^i) \in \mathbb{C}[\![\, \alpha_i \,]\!] \rangle ::$$
$$B = \langle \cup\, i : 1 \le i \le n :: \beta^i \rangle \wedge$$
$$\sigma' = (\sigma - \langle \cup\, i : 1 \le i \le n \wedge \Theta^i(B) :: \sigma^i_{del} \rangle) + \langle \cup\, i : 1 \le i \le n \wedge \Theta^i(B) :: \sigma^i_{add} \rangle$$
$$)$$

In the next section we define the actual Swarm notation system and use the fact that its semantics are consistent with the model shown above.

4. A Notation System: Swarm

As mentioned above, the state of a Swarm program is captured by a dataspace containing data-tuples, transactions, and coupling relation entries (henceforth referred to as synchrony relation entries as the former implicitly defines the latter). Data-tuples and transactions have the form

type_name(sequence_of_values)

as in the transaction *Search(i, c, a)* or the data tuple *leader(pid)*. For each transaction type there is a transaction definition describing the computation performed by any transaction of this type. A transaction definition consists of a finite list of subtransactions separated by the symbol ||. Each subtransaction is semantically an action in the model described in the previous sections, and has a syntax of the form

variable_list : Θ , query $\rightarrow$ action

The *query* part of a subtransaction is an arbitrary predicate which may involve testing for the presence or absence of data-tuples, transactions, and synchrony relation entries in the dataspace. Commas are used inside the query as shorthand for the logical *and* ($\wedge$). A successful query binds the variables in the *variable_list* to values that satisfy the query. All the variables in the variable list are existentially quantified by definition and their scope extends to the action part of the subtransaction and no further. The consensus function Θ may be omitted (for regular queries), or can be any of **TRUE**, **AND**, **NAND**, **OR**, or **NOR** (for special queries). Each regular query contributes *{true}* (in case of success) or *{false}* (in case of failure) to the argument of Θ. All special queries contribute the empty set. All the subtransactions for which Θ (if present) and the query evaluate to true perform their respective *action* part. Each action can be a null operation (**skip**) or a list of deletions and insertions to be performed on the dataspace. All deletions are performed before any insertion. All three kinds of entities can be inserted (by naming the entity to insert) but only data-tuples and synchrony relation entries can be deleted (by using the name of the entity to delete followed by a †). We do not permit the deletion of transactions to ensure fairness.

Two transactions *Count(2, 4)* and *Count(0, 0)* can be explicitly synchronized by introducing a synchrony relation entry of the form

Count(2, 4)~Count(0, 0)

in the dataspace. The synchronization relation entry can be removed by using the syntax

(Count(2, 4)~Count(0, 0))†

If both transactions are present in the dataspace, they are part of some synchronic group and are executed synchronously. If one or the other is missing, the transaction which is present executes by itself—there is no blocking in Swarm. If several transactions of type Count existing in the dataspace are coupled to Count(0,0), they are part of the same synchronic group whether or not Count(0,0) is present at the time. The execution of a synchronic group takes place whenever any one of its members is selected for execution. The selection is weakly fair, i.e., all transactions are eventually selected. The execution involves four phases: the evaluation of the *query*,

the broadcast of its outcome and the evaluation of Θ, the deletion of dataspace entities as described in the actions following successful queries, and the insertion of entities from the same actions.

A complete Swarm program consists of a program header defining the parameters of the program, macro definitions, data and transaction type declarations, and an initialization specifying the initial program state. Most of the notation is straightforward except for the generator construct which is used to specify compactly groups of objects, statements or dataspace entities. For instance, the dataspace may be initialized to contain the elements of index between *1* and *N* of the array *A* using the generator

$$[\ i : 1 \le i \le N :: A(i,f(i))\ ]$$

where *f* is a function in the argument list of the program. Although in Swarm there are many other uses of the generator, we will ignore them since this is the only form employed in the leader election program shown in the next section.

5. An Example: Leader Election

Leader election is a classic problem in which a process must be selected to act as a leader for a set of concurrent processes. This situation can arise when some distributed computation must be supervised by a distinguished process whose identity is not known at the start of the computation. Numerous solutions have been found for the specific case of the processes being distributed in a ring [4, 8, 9].

We assume that each process has a unique identifier from a large set of positive integers. Let *m* be the largest such identifier, and let *N* be the number of processes. Since it is possible to reach any number *m* in *O(log m)* using binary search, it should be possible to elect a leader in no more than this amount of time just by having each process count until the counter is equal to the *smallest* identifier among all the processes. One solution is to divide the counting in two phases. The first phase corresponds to an exponential counting of powers of 2 by each process until the count exceeds the identifier of at least one process. The second phase corresponds to a simple binary search until the smallest identifier is reached. The algorithm is given in Figure 1.

```
program Leader-election(m, N, ID : integer(m), integer(N), ID : integer → integer)
tuple types
    [ i : 0 ≤ i ≤ m :: leader(i) ]
transaction types
    [ pid, c, a : 0 ≤ pid ≤ m , integer(c), integer(a) ::
        Count(pid, c) ≡
                pid > c → skip
            ||  TRUE → (Count(pid, c)~Count(0, 0))†
            ||  AND → Count(pid, c*2), Count(pid, c*2)~Count(0, 0)
            ||  NAND, pid ≤ c → Search(pid, c, c/4), Search(pid, c, c/4)~Search(0,0,0)
            ||  NAND, pid > c → OtherWork(pid);
        Search(pid, c, a) ≡
                pid ≤ c → skip
            ||  TRUE → (Search(pid, c, a)~Search(0,0,0))†
            ||  OR, pid>c → OtherWork(pid)
            ||  OR, pid ≤ c, a > 0 → Search(pid, c-a, a/2), Search(pid, c-a, a/2)~Search(0,0,0)
            ||  NOR, a > 0 → Search(pid, c+a, a/2) , Search(pid, c+a, a/2)~Search(0,0,0)
            ||  TRUE, a = 0, c = pid → leader(pid);
        OtherWork(pid) ≡ <work to be performed asynchronously by losers>
    ]
initialization
    [ i : 0 < i ≤ N :: Count(ID(i),1), Count(ID(i),1)~Count(0,0) ]
end
```

Figure 1: Leader election using built-in consensus

Each process is initially represented by a *Count(pid,c)* transaction, where *pid* is the identifier of the process and *c* is the power of 2 the process has reached so far. A "dummy" transaction *Count(0,0)* serves as a synchronization reference for all the created *Count* transactions. The dummy is never created and is used for clarity purposes only. All the *Count(pid,c)* transactions double their counters synchronously until they "learn", using the special queries **AND** and **NAND**, that some process identifier has been exceeded. All the processes whose identifiers are still greater than the common counter are freed from the election and can execute other computations asynchronously (*OtherWork(pid)*). The remaining processes change representation by becoming *Search(pid,c,a)*

transactions which execute the second phase of the algorithm. The parameters *pid*, *c*, and *a* are, respectively, the identifier of the process, the value of the counter, and the decrement or increment to the counter after the next synchronous step. As with the *Count* transactions, we use a dummy *Search(0,0,0)* as a synchronization reference. The synchronous execution of the *Search* transactions continues until the increment value equals zero, in which case the counter is equal to the smaller identifier which is declared the leader. Throughout the search, any transaction that "learns" that it cannot be the leader leaves the election and proceeds asynchronously on its own. A graphical representation of the computation is shown in Figure 2.

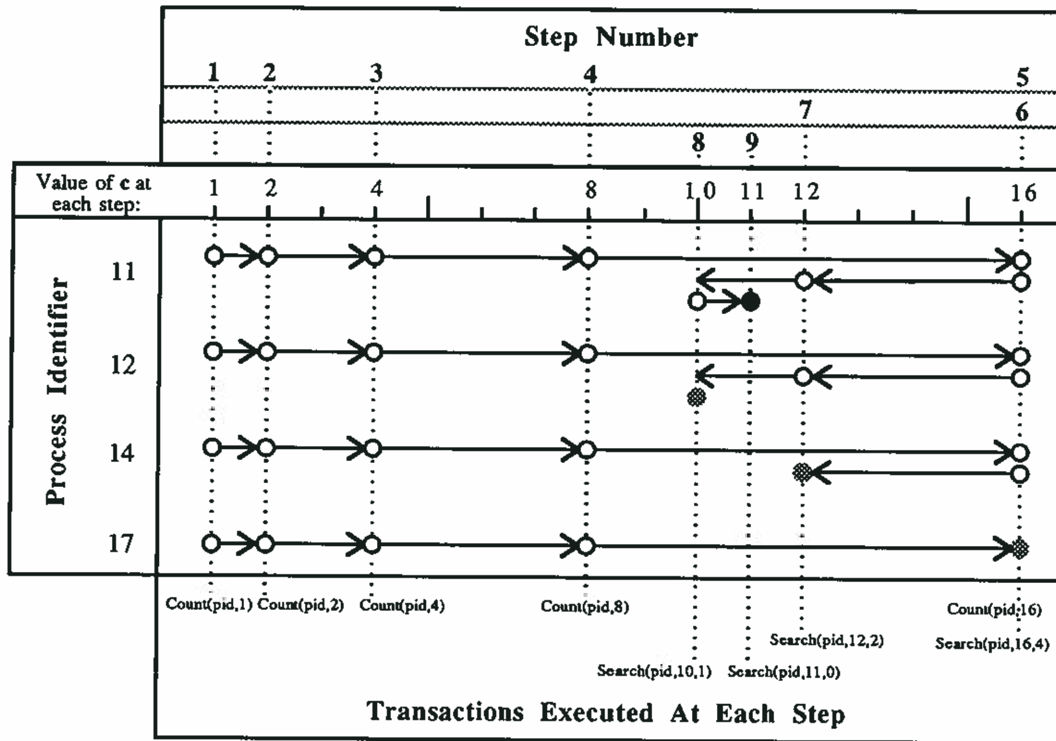

Figure. 2: Computation during election of the leader

The number of steps needed to reach the state where only the leader is in a synchronic group alone is $2*log(s)$ where s is the smallest identifier, compared to $O(N2^S)$ in [8]. The improved result is due to the use of the built-in consensus capability supplied by the special queries.

6. Discussion

Although we employed Swarm to express the algorithm above, its usage is strictly incidental to the purposes of this paper. The real issue at hand is whether or not *dynamic synchrony (DS)* is helpful or necessary to the development and/or modeling of systems that involve concurrent processing. The example shown provides indications that DS fosters elegant and efficient solutions to certain problems. Other problems we have recently studied, not included in this paper due to space considerations, provide additional evidence to support this viewpoint and demonstrate that dynamic synchrony is needed to model certain kinds of distributed architectures. In the remainder of this section we summarize the evidence accumulated to date on behalf of dynamic synchrony.

High-level programming. The key to an elegant solution to any programming problem is the choice of a proper level of abstraction. Most languages tend to support either fine- or large-grained concurrency but not both. DS allows essentially fine-grained languages to construct, at run-time, large-grained atomic operations in direct response to the needs of a specific computing problem. One can reasonably expect that whenever several simple atomic actions are combined into a single larger one, some reduction in the effort to understand and verify the program ought to take place because the number of possible interleavings of actions one must consider is decreased.

Normally one does not associate synchrony with consensus. This connection is a feature particular to Swarm. Nevertheless, it has been our experience that there is very little payoff to using DS unless the participating actions can take decisions based upon some knowledge about the success and failure of other actions participating in the synchronous execution. This capability is heavily used in the leader election algorithm in order to allow processes that lose the election to drop out of the competition.

Finally, it must be noted that these attractive features of DS would be of limited value in the absence of methods that allow for formal verification of programs using synchrony. Fortunately, the work on Swarm led to the development of a UNITY-like proof logic that covers dynamic synchronic groups and built-in consensus [6, 14].

Modeling and programming unconventional architectures. Throughout this paper we treated DS as a programming language feature and argued it could be helpful in the development of concurrent programs.

We believe, however, that in the development of software targeted to heterogeneous, reconfigurable, and/or specialized architectures the ability to specify and reason about various forms of synchrony is necessary, not merely convenient. Current trends toward multimedia interfaces clearly require very efficient synchronous processing of video input and output. One way to accomplish this is to envision an architecture which permits dynamic configuration of synchronous pipelines out of a set of specialized video processing components. Finally, specialized applications such as distributed simulation could greatly benefit from architectures that have some hard-wired consensus-reaching mechanisms. Such mechanisms would allow, for instance, individual processors to determine with very little overhead when to update the local simulation clocks. All these architectural choices can be specified in Swarm and have contributed to shaping our thoughts about dynamic synchrony.

7. Conclusions

In this paper we presented a formal model which views synchrony as a relation between atomic actions, a relation which may evolve in time. The appropriateness and generality of the model was demonstrated by instantiating it for several languages and models that exhibit various degrees and styles of synchrony. The possibility that programmers could be allowed to explicitly specify and reason about dynamic synchrony was explored via an example which provides ample evidence that the study of dynamic synchrony holds significant promise and, as such, it ought to be the subject of a systematic formal and pragmatic evaluation.

Acknowledgments

This paper is based on work supported by the National Science Foundation under Grant No. CCR-9015677. The Government has certain rights in this material. The authors thank Kenneth Cox for his constructive criticisms and review of this paper.

A. References

1. "Ada® Programming Language," American National Standards Institute, Inc. (1983).

2. Back, R. J. R., and Kurki-Suonio, R., "Distributed Cooperation with Action Systems," *CM Transactions on Programming Languages and Systems* **10**(4), pp. 513-554 (1988).

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

4. Chang, E., and Roberts, R., "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configuration of Processes," *Communication of the ACM* **22**, pp. 281-283 (1979).

5. Charlesworth, A., "The Multiway Rendezvous," *ACM Transactions on Programming Languages and Systems* **9**(2), pp. 350-366 (1987).

6. Cunningham, H. C., and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems* **1**(3), pp. 365-376 (1990).

7. Evangelist, M., Francez, N., and Katz, S., "Multiparty Interactions for Interprocess Communication and Synchronization," *IEEE Transactions on Software Engineering* **15**(11), pp. 1417-1426 (1989).

8. Frederickson, G. N., and Lynch, N., "Electing a Leader in a Synchronous Ring," *Journal of the ACM* **34**(1), pp. 98-115 (1987).

9. Hirschberg, D. S., and Sinclair, J. B., "Decentralized Extrema-Finding in Circular Configurations of Processes," *Communications of the ACM* **23**, pp. 627-628 (1980).

10. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* **21**(8), pp. 666-677 (1978).

11. Lynch, N. A., and Tuttle, M. R., "An Introduction to Input/Output Automata," *CWI Quaterly* **2**(3), pp. 219-246 (1989).

12. Ramesh, S., and Mehndiratta, S. L., "A Methodology for Developing Distributed Programs," *IEEE Transactions on Software Engineering* **SE-13**(8), pp. 967-976 (1987).

13. Roman, G.-C., and Cunningham, H. C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering* **16**(12), pp. 1361-1373 (1990).

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