A Structured Approach to Parallel Programming: Methodology and Models

UF CISE Technical Report 98-023 ⋆

Berna L. Massingill

University of Florida, P.O. Box 116120, Gainesville, FL 32611
blm@cise.ufl.edu

Abstract. Parallel programming continues to be difficult, despite substantial and ongoing research aimed at making it tractable. Especially dismaying is the gulf between theory and the practical programming. We propose a structured approach to developing parallel programs for problems whose specifications are like those of sequential programs, such that much of the work of development, reasoning, and testing and debugging can be done using familiar sequential techniques and tools. The approach takes the form of a simple model of parallel programming, a methodology for transforming programs in this model into programs for parallel machines based on the ideas of semantics-preserving transformations and programming archetypes (patterns), and an underlying operational model providing a unified framework for reasoning about those transformations that are difficult or impossible to reason about using sequential techniques. This combination of a relatively accessible programming methodology and a sound theoretical framework to some extent bridges the gulf between theory and practical programming. This paper sketches our methodology and presents our programming model and its supporting framework in some detail.

1 Introduction

Despite the past and ongoing efforts of many researchers, parallel programming continues to be difficult, with a persistent and dismaying gulf between theory and practical programming. We propose a structured approach to developing parallel programs for the class of problems whose specifications are like those usually given for sequential programs, in which the specification describes initial states for which the program must terminate and the relation between initial and final states. Our approach allows much of the work of development, reasoning, and testing and debugging to be done using familiar sequential techniques and tools; it takes the form of a simple model of parallel programming, a methodology for transforming programs in this model into programs for parallel machines

⋆ This work was supported by funding from the Air Force Office of Scientific Research (AFOSR) and the National Science Foundation (NSF).

based on the ideas of semantics-preserving transformations and programming archetypes (patterns), and an underlying operational model providing a unified framework for reasoning about those transformations that are difficult or impossible to reason about using sequential techniques.

By combining a relatively accessible programming methodology with a sound theoretical framework, our approach to some extent bridges the gap between theory and practical programming. The transformations we propose are in many cases formalized versions of what programmers and compilers typically do in practice to "parallelize" sequential code, but we provide a framework for formally proving their correctness (either by standard sequential techniques or by using our operational model). Our operational model is sufficiently general to support proofs of transformations between markedly different programming models (sequential, shared-memory, and distributed-memory with message-passing). It is sufficiently abstract to permit a fair degree of rigor, but simple enough to be relatively accessible, and applicable to a range of programming notations.

This paper describes our programming methodology and presents our programming model and its supporting framework in some detail. It also sketches briefly how the model applies in the context of particular programming notations.

2 Our programming model and methodology

Our programming model comprises a primary model and two subsidiary models, and is designed to support a programming methodology based on stepwise refinement and the reuse where possible of the techniques and tools of sequential programming. This section gives an overview of our model and methodology.

2.1 The arb model: parallel composition with sequential semantics

Our primary programming model, which we call the ***arb** model*, is simply the standard sequential model (as defined by Dijkstra [14, 15], Gries [18], and others) extended to include parallel compositions of groups of program elements whose parallel composition is equivalent to their sequential composition. The name (**arb**) is derived from UC (Unity C) [5] and is intended to connote that such groups of program elements may be interleaved in any arbitrary fashion without changing the result. We define a property we call ***arb**-compatibility*, and we show that if a group of program elements is **arb**-compatible, their parallel composition is semantically equivalent to their sequential composition; we call such compositions ***arb** compositions*. Since **arb**-model programs can be interpreted as sequential programs, the extensive body of tools and techniques applicable to sequential programs is applicable to them. In particular, their correctness can be demonstrated formally by using sequential methods, they can be refined by sequential semantics-preserving transformations, and they can be executed sequentially for testing and debugging.

2.2 Transformations from the arb model to practical parallel languages

Because the **arb** composition of **arb**-compatible elements can also be interpreted as parallel composition, **arb**-model programs can be executed as parallel programs. Such programs may not make effective use of typical parallel architectures, however, so our methodology includes techniques for improving their efficiency while maintaining correctness. We define two subsidiary programming models that abstract key features of two classes of parallel architectures: the ***par** model* for shared-memory (single-address-space) architectures, and the *subset **par** model* for distributed-memory (multiple-address-space) architectures. We then develop semantics-preserving transformations to convert **arb**-model programs into programs in one of these subsidiary models. Intermediate stages in this process are usually **arb**-model programs, so the transformations can make use of sequential refinement techniques, and the programs can be executed sequentially. Finally, we indicate how the **par** model can be mapped to practical programming languages for shared-memory architectures and the subset **par** model to practical programming languages for distributed-memory–message-passing architectures. Together, these groups of transformations provide a semantics-preserving path from the original **arb**-model program to a program in a practical programming language. Figure 1 illustrates this overall scheme.

2.3 Supporting framework for proving transformations correct

Some of the transformations indicated in Figure 1 — those within the **arb** model — can be proved correct using the techniques of sequential stepwise refinement (as defined by Gries [18], Hoare [20], and others). Others — those between our different programming models, or from one of our models to a practical programming language — require a different approach. We therefore define an operational model based on viewing programs as state-transition systems, give definitions of our programming models in terms of this underlying operational model, and use it to prove the correctness of those transformations for which sequential techniques are inappropriate.

2.4 Programming archetypes

An additional important element of our approach, though not one that will be addressed in this paper, is that we envision the transformation process just described as being guided by *parallel programming archetypes*, by which we mean abstractions that capture the commonality of classes of programs, much like the design patterns [17] of the object-oriented world. We envision application developers choosing from a range of archetypes, each representing a class of programs with common features and providing a class-specific parallelization strategy (i.e., a pattern for the shared-memory or distributed-memory program to be ultimately produced) together with a collection of class-specific transformations and a code library of communication or other operations that encapsulate the

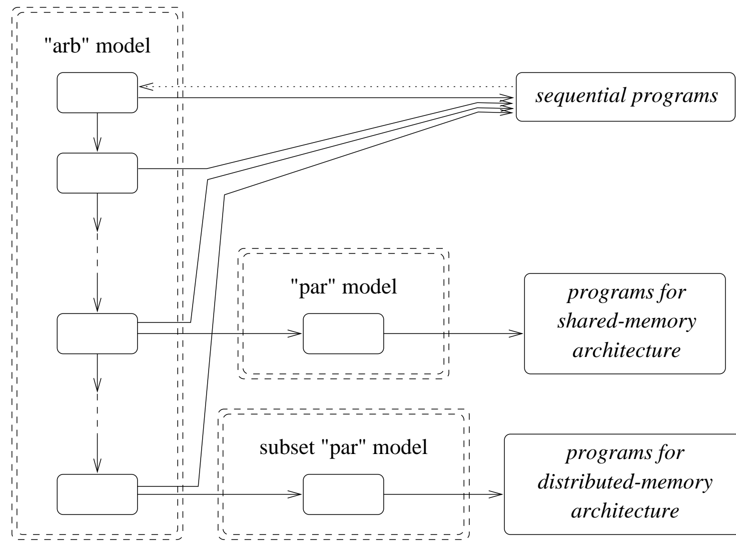

Fig. 1. Overview of programming models and transformation process. Solid-bordered boxes indicate programs in the various models; arrows indicate semantics-preserving transformations. A dashed arrow runs from the box denoting a sequential program to a box denoting an **arb**-model programs because it is sometimes appropriate and feasible to derive an **arb**-model program from an existing sequential program (by replacing sequential compositions of **arb**-compatible elements with **arb** compositions of the same elements).

details of the parallel programs. Archetypes are described in more detail in [8] and [30].

2.5 Program development using our methodology

We can then employ the following approach to program development, with all steps guided by an archetype-specific parallelization strategy and supported by a collection of archetype-appropriate already-proved-correct transformations.

Development of initial program. The application developer begins by developing a correct program using sequential constructors and parallel composition (||), but ensuring that all groups of elements composed in parallel are **arb**-compatible. We call such a program an ***arb**-model program*, and it can be interpreted as either a sequential program or a parallel program, with identical meaning. Correctness of this program can be established using techniques for establishing correctness of a sequential program.

Sequential-to-sequential refinement. The developer then begins the process of refining the program into one suitable for the target architecture. During the initial stages of this process, the program is viewed as a sequential program and operated on with sequential refinement techniques, which are well-defined and well-understood. A collection of representative transformations (refinement steps) is presented in [30]. (Appendix C sketches a few of them.) In refining a sequential composition whose elements are **arb**-compatible, care is taken to preserve their **arb**-compatibility. The result is a program that refines the original program and can also be interpreted as either a sequential or a parallel program, with identical meaning. The end product of this refinement should be a program that can then be transformed into an efficient program in the **par** model (for a shared-memory target) or the subset **par** model (for a distributed-memory target).

Sequential-to-parallel refinement. The developer next transforms (refines) the refined **arb**-model program into a **par**-model or subset-**par**-model program.

Translation for target platform. Finally, the developer translates the **par**-model or subset-**par**-model program into the desired parallel language for execution on the target platform. This step, like the others, is guided by semantics-preserving rules for mapping one of our programming models into the constructs of a particular parallel language.

3 The arb model

As discussed in Section 2, the heart of our approach is identifying groups of program elements that have the useful property that their parallel composition

is semantically equivalent to their sequential composition. We call such a group of program elements **arb**-compatible.

In this section, we first present our operational model for parallel programs, the model we will use for reasoning about programs and program transformations that are not amenable to strictly sequential reasoning techniques. We then define a notion of **arb**-compatibility, such that the parallel composition of a group of **arb**-compatible program elements is semantically equivalent to its sequential composition. We then identify restrictions on groups of program elements that are sufficient to guarantee their **arb**-compatibility, and we present some properties of parallel compositions of **arb**-compatible elements. Finally, we sketch how these ideas apply in the context of programming notations and briefly discuss executing **arb**-model programs sequentially and in parallel.

It is worth observing at this point that the ideas behind the programming model are not tied to any particular programming notation but should apply to any imperative programming notation. We present definitions and theorems for our programming models in a notation based on that of Dijkstra's guarded-command language, since it is a simple and compact notation that makes for readable definitions and theorems. However, we present examples of applying the definitions and theorems in a notation based on Fortran 90, in order to take advantage of Fortran 90's wider range of convenient constructs (e.g., arrays and `DO` loops) and to indicate how our ideas apply in the context of a practical programming notation.

3.1 Overview of program semantics and operational model

We define programs in such a way that a program describes a state-transition system, and show how to define program computations, sequential and parallel composition, and program refinement in terms of this definition. In this paper we present this material with a minimum of mathematical notation and only brief sketches of most proofs; a more formal treatment of the material, including more complete proofs, appears in [30].

Treating programs as state-transition systems is not a new approach; it has been used in work such as Chandy and Misra [9], Lynch and Tuttle [24], Lamport [23], Manna and Pnueli [26], and Pnueli [34] to reason about both parallel and sequential programs. The basic notions of a state-transition system — a set of states together with a set of transitions between them, representable as a directed graph with states for vertices and transitions for edges — are perhaps more helpful in reasoning about parallel programs, particularly when program specifications describe ongoing behavior (e.g., safety and progress properties) rather than relations between initial and final states, but they are also applicable to sequential programs. Our operational model builds on this basic view of program execution, presented in a way specifically aimed at facilitating the stating and proving of the main theorems of this section (that for groups of program elements meeting stated criteria, their parallel and sequential compositions are semantically equivalent) and subsequent sections.

3.2 Definitions

Definition 3.1 (Program).

We define a program P as a 6-tuple $(V, L, InitL, A, PV, PA)$, where

- V is a finite set of typed variables. V defines a state space in the state-transition system; that is, a state is given by the values of the variables in V. In our semantics, distinct program variables denote distinct atomic data objects; aliasing is not allowed.
- $L \subseteq V$ represents the *local variables* of P. These variables are distinguished from the other variables of P in two ways: (i) The initial states of P are given in terms of their values, and (ii) they are invisible outside P — that is, they may not appear in a specification for P, and they may not be accessed by other programs composed with P, either in sequence or in parallel.
- $InitL$ is an assignment of values to the variables of L, representing their initial values.
- A is a finite set of *program actions*. A program action describes a relation between states of its input variables (those variables in V that affect its behavior, either in the sense of determining from which states it can be executed or in the sense of determining the effects of its execution) and states of its output variables (those variables whose value can be affected by its execution). Thus, a program action is a triple (I_a, O_a, R_a) in which
  - $I_a \subseteq V$ represents the input variables of A.
  - $O_a \subseteq V$ represents the output variables of A.
  - R_a is a relation between I_a-tuples and O_a-tuples.
- $PV \subseteq V$ are *protocol variables* that can be modified only by *protocol actions* (elements of PA). (That is, if v is a protocol variable, and $a = (I_a, O_a, R_a)$ is an action such that $v \in O_a$, a must be a protocol action.) Such variables and actions are not needed in this section but are useful in defining the synchronization mechanisms of Section 4 and Section 5; the requirement that protocol variables be modified only by protocol actions simplifies the task of defining such mechanisms. Observe that variables in PV can include both local and non-local variables.
- $PA \subseteq A$ are *protocol actions*. Only protocol actions may modify protocol variables. (Protocol actions may, however, modify non-protocol variables.)

A program action $a = (I_a, O_a, R_a)$ defines a set of state transitions, each of which we write in the form $s \stackrel{a}{\to} s'$, as follows: $s \stackrel{a}{\to} s'$ if the pair (i, o), where i is a tuple representing the values of the variables in I_a in state s and o is a tuple representing the values of the variables in O_a in state s', is an element of relation R_a.

Observe that we can also define a program action based on its set of state transitions, by inferring the required I_a, O_a, and R_a.

□

Appendix A presents examples of defining the commands of a programming notation (Dijkstra's guarded-command language [13, 15]) in terms of our model.

Definition 3.2 (Initial states).

For program P, s is an *initial state* of P if, in s, the values of the local variables of P have the values given in *InitL*.

□

Definition 3.3 (Enabled).

For action a and state s of program P, we say that a is *enabled* in s exactly when there exists program state s' such that $s \stackrel{a}{\to} s'$.

□

Definition 3.4 (Computation).

If $P = (V, L, \mathit{InitL}, A, PV, PA)$, a *computation* of P is a pair

$$C = (s_0, \langle j : 1 \leq j \leq N : (a_j, s_j) \rangle)$$

in which

- s_0 is an initial state of P.
- $\langle j : 1 \leq j \leq N : (a_j, s_j) \rangle$ is a sequence of pairs in which each a_j is a program action of P, and for all j, $s_{j-1} \stackrel{a_j}{\to} s_j$. We call these pairs the state transitions of C, and the sequence of actions a_j the actions of C. N can be a non-negative integer or ∞. In the former case, we say that C is a finite or terminating computation with length $N + 1$ and final state s_N. In the latter case, we say that C is an infinite or nonterminating computation.
- If C is infinite, the sequence $\langle j : 1 \leq j : (a_j, s_j) \rangle$ satisfies the following fairness requirement: If, for some state s_j and program action a, a is enabled in s_j, then eventually either a occurs in C or a ceases to be enabled.

□

Definition 3.5 (Terminal state).

We say that state s of program P is a *terminal state* of P exactly when there are no actions of P enabled in s.

□

Definition 3.6 (Maximal computation).

We say that a computation of C of P is a *maximal computation* exactly when either (i) C is infinite or (ii) C is finite and ends in a terminal state.

□

Definition 3.7 (Affects).

For predicate q and variable $v \in V$, we say that v *affects* q exactly when there exist states s and s', identical except for the value of v, such that $q.s \neq q.s'$. For expression E and variable $v \in V$, we say that v affects E exactly when there exists value k for E such that v affects the predicate $(E = k)$.

□

3.3 Specifications and program refinement

The usual meaning of "program P is refined by program P'" is that program P' meets any specification met by P. We will confine ourselves to specifications that describe a program's behavior in terms of initial and final states, giving (i) a set of initial states s such that the program is guaranteed to terminate if started in s, and (ii) the relation, for terminating computations, between initial and final states. An example of such a specification is a Hoare total-correctness triple [20]. In terms of our model, initial and final states correspond to assignments of values to the program's variables; we make the additional restriction that specifications do not mention a program's local variables L. We make this restriction because otherwise program equivalence can depend on internal behavior (as reflected in the values of local variables), which is not the intended meaning of equivalence. We write $P \sqsubseteq P'$ to denote that P is refined by P'; if $P \sqsubseteq P'$ and $P' \sqsubseteq P$, we say that P and P' are equivalent, and we write $P \sim P'$.

Definition 3.8 (Equivalence of computations).

For programs P_1 and P_2 and a set of typed variables V such that $V \subseteq V_1$ and $V \subseteq V_2$ and for every v in V, v has the same type in all three sets (V, V_1, and V_2), we say that computations C_1 of P_1 and C_2 of P_2 are *equivalent with respect to V* exactly when:

- For every v in V, the value of v in the initial state of C_1 is the same as its value in the initial state of C_2.
- Either (i) both C_1 and C_2 are infinite, or (ii) both are finite, and for every v in V, the value of v in the final state of C_1 is the same as its value in the final state of C_2.

□

We can now give a sufficient condition for showing that $P_1 \sqsubseteq P_2$ in our semantics.

Theorem 3.9 (Refinement in terms of equivalent computations).

For P_1 and P_2 with $(V_1 \setminus L_1) \subseteq (V_2 \setminus L_2)$ (where $\setminus$ denotes set difference), $P_1 \sqsubseteq P_2$ when for every maximal computation C_2 of P_2 there is a maximal computation C_1 of P_1 such that C_1 is equivalent to C_2 with respect to $(V_1 \setminus L_1)$.

□

Proof of Theorem 3.9.

This follows immediately from Definition 3.8, the usual definition of refinement, and our restriction that program specifications not mention local variables.

□

3.4 Program composition

We now present definitions of sequential and parallel composition in terms of our model. First we need some restrictions to ensure that the programs to be composed are compatible — that is, that it makes sense to compose them:

Definition 3.10 (Composability of programs).

We say that a set of programs $P_1, \ldots, P_N$ *can be composed* exactly when

- any variable that appears in more than one program has the same type in all the programs in which it appears (and if it is a protocol variable in one program, it is a protocol variable in all programs in which it appears),
- any action that appears in more than one program is defined in the same way in all the programs in which it appears, and
- different programs do not have local variables in common.

□

Sequential composition

The usual meaning of sequential composition is this: A maximal computation of $P_1; P_2$ is a maximal computation C_1 of P_1 followed (if C_1 is finite) by a maximal computation C_2 of P_2, with the obvious generalization to more than two programs. We can give a definition with this meaning in terms of our model by introducing additional local variables $En_1, \ldots, En_N$ that ensure that things happen in the proper sequence, as follows: Actions from program P_j can execute only when En_j is *true*. En_1 is set to *true* at the start of the computation, and then as each P_j terminates it sets En_j to *false* and En_{j+1} to *true*, thus ensuring the desired behavior.

Definition 3.11 (Sequential composition).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 3.10), we define their sequential composition $(P_1; \ldots; P_N) = (V, L, InitL, A, PA, PV)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V:
  En_P is *true* in the initial state of the sequential composition and *false* thereafter.
  For all j, En_j is *true* during (and only during) the part of the computation corresponding to execution of P_j.
- $InitL$ is defined thus: The initial value of En_P is *true*. For all j, the initial value of En_j is *false*, and the initial values of variables in L_j are those given by $InitL_j$.
- A consists of the following types of actions:
  - Actions corresponding to actions in A_j, for some j: For $a \in A_j$, we define a' identical to a except that a' is enabled only when $En_j = true$.
  - Actions that accomplish the transitions between components of the composition:
    Initial action a_{T_0} takes any initial state s, with $En_P = true$, to a state s' identical except that $En_P = false$ and $En_1 = true$. s' is thus an initial state of P_1.
    For j with $1 \leq j < N$, action a_{T_j} takes any terminal state s of P_j, with $En_j = true$, to a state s' identical except that $En_j = false$ and $En_{j+1} = true$. s' is thus an initial state of P_{j+1}.
    Final action a_{T_N} takes any terminal state s of P_N, with $En_N = true$, to a state s' identical except that $En_N = false$. s' is thus a terminal state of the sequential composition.
- $PV = PV_1 \cup \ldots \cup PV_N$.
- PA contains exactly those actions a' derived (as described above) from the actions a of $PA_1 \cup \ldots \cup PA_N$.

□

Parallel composition

The usual meaning of parallel composition is this: A computation of $P_1 || P_2$ defines two threads of control, one each for P_1 and P_2. Initiating the composition corresponds to starting both threads; execution of the composition corresponds to an interleaving of actions from both components; and the composition is understood to terminate when both components have terminated. We can give a definition with this meaning in terms of our model by introducing additional local variables that ensure that the composition terminates when all of its components terminate, as follows: As for sequential composition, we introduce additional

local variables $En_1, \ldots, En_N$ such that actions from program P_j can execute only when En_j is *true*. For parallel composition, however, all of the En_j's are set to *true* at the start of the computation, so computation is an interleaving of actions from the P_j's. As each P_j terminates, it sets the corresponding En_j to *false*; when all are *false*, the composition has terminated. Observe that the definitions of parallel and sequential composition are almost identical; this greatly facilitates the proofs of Lemma 3.17 and Lemma 3.18.

Definition 3.12 (Parallel composition).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 3.10), we define their parallel composition $(P_1 || \ldots || P_N) = (V, L, InitL, A, PV, PA)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V:
  En_P is *true* in the initial state of the parallel composition and *false* thereafter.
  For all j, En_j is *true* until the part of the composition corresponding to P_j has terminated.
- $InitL$ is defined thus: The initial value of En_P is *true*. For all j, the initial value of En_j is *false*, and the initial values of variables in L_j are those given by $InitL_j$.
- A consists of the following types of actions:
  - Actions corresponding to actions in A_j, for some j: For $a \in A_j$, we define a' identical to a except that a' is enabled only when En_j is *true*.
  - Actions that correspond to the initiation and termination of the components of the composition:
    Initial action a_{T_0} takes any initial state s, with $En_P = true$, to a state s' identical except that $En_j = true$ for all j. s' is thus an initial state of P_j, for all j.
    For j with $1 \leq j \leq N$, action a_{T_j} takes any terminal state s of P_j, with $En_j = true$, to a state s' identical except that $En_j = false$. A terminating computation of P contains one execution of each a_{T_j}; after execution of a_{T_j} for all j, the resulting state s' is a terminal state of the parallel composition.
- $PV = PV_1 \cup \ldots \cup PV_N$.
- PA contains exactly those actions a' derived (as described above) from the actions a of $PA_1 \cup \ldots \cup PA_N$.

□

3.5 arb-compatibility

We now turn our attention to defining sufficient conditions for a group of programs $P_1, \ldots, P_N$ to have the property we want, namely:

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \ .$$

We first define a key property of pairs of program actions; we can then define the desired condition and show that it guarantees the property of interest.

Definition 3.13 (Commutativity of actions).

Actions a and b of program P are said to *commute* exactly when the following two conditions hold:

- Execution of b does not affect (in the sense of Definition 3.7) whether a is enabled, and vice versa.
- It is possible to reach s_2 from s_1 by first executing a and then executing b exactly when it is also possible to reach s_2 from s_1 by first executing b and then executing a, as illustrated by Figure 2. then executing a. (Observe that if a and b are nondeterministic, there may be more than one such state s_2.)

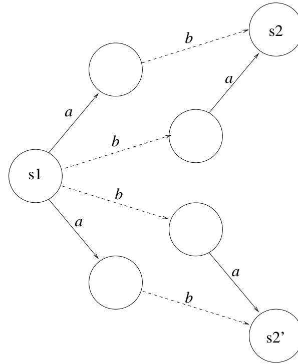

Fig. 2. Commutativity of actions a and b. Observe that a and b are nondeterministic, but the graph has the property that if we can reach a state (s_2 or s_2') by executing first a and then b, then we can reach the same state by first executing b and then a, and vice versa.

(That is, a and b commute exactly when they have the diamond property [10, 25].)

□

*Definition 3.14 (**arb**-compatible).*

Programs $P_1, \ldots, P_N$ are ***arb**-compatible* exactly when they can be composed (Definition 3.10) and any action in one program commutes (Definition 3.13) with any action in another program.

□

Theorem 3.15 (Parallel ∼ sequential for **arb***-compatible programs).*

If $P_1, \ldots, P_N$ are **arb**-compatible, then

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \enspace .$$

□

Proof of Theorem 3.15.

We write $P_P = (P_1 || \ldots || P_N)$ and $P_S = (P_1; \ldots; P_N)$. From Definition 3.11 and Definition 3.12,

$$(V_P = V_S) \ \wedge \ (L_P = L_S) \ \wedge \ (InitL_P = InitL_S) \ \wedge \ (PV_P = PV_S) \\ \wedge \ (PA_P = PA_S) \enspace ,$$

so we write $P_P = (V, L, InitL, A_P, PV, PA)$ and $P_S = (V, L, InitL, A_S, PV, PA)$. We proceed as follows:

- We first show (Lemma 3.17) that for every maximal computation C_S of P_S there is a maximal computation C_P of P_P with C_S equivalent to C_P with respect to $V \setminus L$. From Theorem 3.9, this establishes that $P_P \sqsubseteq P_S$.
- We then show (Lemma 3.18) the converse: that for every maximal computation C_P of P_P there is a maximal computation C_S of P_S with C_P equivalent to C_S with respect to $V \setminus L$. From Theorem 3.9, this establishes that $P_S \sqsubseteq P_P$.
- We then conclude that $P_P \sim P_S$, as desired.

□

Lemma 3.16 (Reordering of computations).

Suppose that $P_1, \ldots, P_N$ are **arb**-compatible and C_P is a finite (not necessarily maximal) computation of $P_P = (P_1 || \ldots || P_N)$ containing a successive pair of transitions $((a, s_n), (b, s_{n+1}))$ such that a and b commute. Then we can construct a computation C'_P of P_P with the same initial and final states as C_P, and the same sequence of transitions, except that the pair $((a, s_n), (b, s_{n+1}))$ has been replaced by the pair $((b, s'_n), (a, s_{n+1}))$.

□

Proof of Lemma 3.16.

This is an obvious consequence of the commutativity (Definition 3.13) of a and b: If $s_{n-1} \stackrel{a}{\rightarrow} s_n$ and $s_n \stackrel{b}{\rightarrow} s_{n+1}$, then there exists a state s'_n such that $s_{n-1} \stackrel{b}{\rightarrow} s'_n$ and $s'_n \stackrel{a}{\rightarrow} s_{n+1}$, so we can construct a computation as described.

□

Lemma 3.17 (Sequential refines parallel).

For P_P and P_S defined as in Theorem 3.15, if C_S is a maximal computation of P_S, there is a maximal computation C_P of P_P with C_S equivalent to C_P with respect to $V \setminus L$.

□

Proof of Lemma 3.17.

The proof of this lemma is straightforward for finite computations: We have defined parallel and sequential composition in such a way that any maximal finite computation of the parallel computation maps to an equivalent maximal computation of the parallel composition.

For nonterminating computations, we can similarly map a computation of the sequential composition to an infinite sequence of transitions of the parallel composition. However, the result may not be a computation of the parallel composition because it may violate the fairness requirement: If P_j fails to terminate, no action of P_{j+1} can occur, even though in the parallel composition there may be actions of P_{j+1} that are enabled. If this is the case, however, we can use the principle behind Lemma 3.16 to transform the unfair sequence of transitions into a fair one.

□

Lemma 3.18 (Parallel refines sequential).

For P_P and P_S defined as in Theorem 3.15, if C_P is a maximal computation of P_P, there is a maximal computation C_S of P_S such that C_S is equivalent to C_P with respect to $V \setminus L$.

□

Proof of Lemma 3.18.

For terminating computations, the proof is straightforward: Given a maximal computation of the parallel composition, we first apply Lemma 3.16 repeatedly to construct an equivalent (also maximal) computation of the parallel composition

in which, for $j < k$, all transitions corresponding to actions of P_j occur before transitions corresponding to actions of P_k. As in the proof of Lemma 3.17, this computation then maps to an equivalent maximal computation of the sequential composition.

For nonterminating computations, we can once again use the principle behind Lemma 3.16 to construct a sequence of transitions (of the parallel composition) in which, for $j < k$, all transitions corresponding to actions of P_j occur before transitions corresponding to actions of P_k. We then map this sequence of transitions to a computation of the sequential composition.

□

3.6 arb composition

For **arb**-compatible programs $P_1, \ldots, P_N$, then, we know that

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \ .$$

To denote this parallel/sequential composition of **arb**-compatible elements, we write $\mathbf{arb}(P_1, \ldots, P_N)$, where

$$\mathbf{arb}(P_1, \ldots, P_N) \sim (P_1 || \ldots || P_N)$$

or equivalently

$$\mathbf{arb}(P_1, \ldots, P_N) \sim (P_1; \ldots; P_N) \ .$$

We refer to this notation as "**arb** composition", although it is not a true composition operator since it is properly applied only to groups of elements that are **arb**-compatible. We regard it as a useful form of syntactic sugar that denotes not only the parallel/sequential composition of $P_1, \ldots, P_N$ but also the fact that $P_1, \ldots, P_N$ are **arb**-compatible. We also define an additional bit of syntactic sugar, $\mathbf{seq}(P_1, \ldots, P_N)$, such that

$$\mathbf{seq}(P_1, \ldots, P_N) \sim (P_1; \ldots; P_N) \ .$$

(We use this notation to improve the readability of nestings of sequential and **arb** composition.)

arb composition has a number of useful properties. It is associative and commutative (proofs given in [30]), and it allows refinement by parts, as the following theorem states.

Theorem 3.19 (Refinement by parts of **arb** *composition).*

We can refine any component of an **arb** composition to obtain a refinement of the whole composition. That is, if $P_1, \ldots, P_N$ are **arb**-compatible, and, for each j, $P_j \sqsubseteq P'_j$, and $P'_1, \ldots, P'_N$ are **arb**-compatible, then

$$\mathbf{arb}(P_1, \ldots, P_N) \sqsubseteq \mathbf{arb}(P'_1, \ldots, P'_N)$$

□

Proof of Theorem 3.19.

This follows from Theorem 3.15 and refinement by parts for sequential programs.

□

This theorem is the justification for our program-development strategy, in which we apply the techniques of sequential stepwise refinement to **arb**-model programs.

3.7 A simpler sufficient condition for arb-compatibility

The definition of **arb**-compatibility given in Definition 3.14 is the most general one that seems to give the desired properties (equivalence of parallel and sequential composition, and associativity and commutativity), but it may be difficult to apply in practice. We therefore give a more-easily-checked sufficient condition for programs $P_1, \ldots, P_N$ to be **arb**-compatible.

Definition 3.20 (Variables read/written by P).

For program P and variable v, we say that v is *read by* P if it is an input variable for some action a of P, and we say that v is *written by* P if it is an output variable for some action a of P.

□

*Theorem 3.21 (**arb**-compatibility and shared variables).*

If programs $P_1, \ldots, P_N$ can be composed (Definition 3.10), and for $j \neq k$, no variable written by P_j is read or written by P_k, then $P_1, \ldots, P_N$ are **arb**-compatible.

□

Proof of Theorem 3.21.

Given programs $P_1, \ldots, P_N$ that satisfy the condition, it suffices to show that any two actions from distinct components P_j and P_k commute. The proof is straightforward; a detailed version appears in [30].

□

3.8 arb composition and programming notations

A key difficulty in applying our methodology for program development is in identifying groups of program elements that are known to be **arb**-compatible. The difficulty is exacerbated by the fact that many programming notations have a notion of program variable that is more difficult to work with than the notion we employ for our formal semantics. In our semantics, variables with distinct names address distinct data objects. In many programming notations, this need not be the case, and the difficulty of detecting situations in which variables with distinct names overlap (aliasing) complicates automatic program optimization and parallelization just as it complicates the application of our methodology. Syntactic restrictions sufficient to guarantee **arb**-compatibility do not seem in general feasible. However, it is feasible to give semantics-based rules that identify, for program P, supersets of the variables read and written by P, and identifying such supersets is sufficient to permit application of Theorem 3.21. In [30] we discuss such rules for two representative programming notations, Dijkstra's guarded-command language [13, 15] and Fortran 90 [22, 1], and present examples of the use of these rules. Defining such rules is fairly straightforward for Dijkstra's guarded-command language, since it is a small and well-understood language. It is less straightforward for a large and complex language such as Fortran 90; giving a formal definition of its semantics is far from trivial. We observe, however, that the well-understood constructs of Dijkstra's guarded-command language have, when deterministic, analogous constructs in Fortran 90 (as in many other practical languages), and that formally-justified results derived in Dijkstra's guarded-command language apply to Fortran 90 programs insofar as the Fortran 90 programs limit themselves to these analogous constructs.

Before presenting examples, we introduce a little additional notation so that we can apply our extensions to the sequential programming model to a representative practical programming notation, Fortran 90. We do this in order to show how our ideas apply in the context of a practical programming notation.

arb *composition.* For **arb**-compatible programs $P_1, \ldots, P_N$, we write their **arb** composition thus:

```
arb
    P_1
    ...
    P_N
end arb
```

seq *composition.* We define an analogous notation for sequential composition, using keywords `seq` and `end seq`, useful in improving the readability of nestings of sequential and **arb** composition.

arball. To allow us to express the **arb** composition of, for example, the iterations of a loop, we define an indexed form of **arb** composition, with syntax

modeled after that of the `FORALL` construct of High Performance Fortran [19], as follows. This notation is syntactic sugar only, and all theorems that apply to **arb** composition apply to **arball** as well.

*Definition 3.22 (***arball***).*

If we have N index variables $i_1, \ldots, i_N$, with corresponding index ranges $i_j_start \leq i_j \leq i_j_end$, and program block P such that P does not modify the value of any of the index variables, then we can define an **arball** composition as follows.

For each tuple $(x_1, \ldots, x_N)$ in the cross product of the index ranges, we define a corresponding program block $P(x_1, \ldots, x_N)$ by replacing index variables $i_1, \ldots, i_N$ with corresponding values $x_1, \ldots, x_N$. If the resulting program blocks are **arb**-compatible, then we write their **arb** composition as follows:

```
arball (i1 = i1_start : i1_end , ..., iN = iN_start : iN_end)
    P(x1, ..., xN)
end arball
```

□

3.9 Examples of arb composition

Composition of sequential blocks. This example composes two sequences, the first assigning to `a` and `b` and the second assigning to `c` and `d`.

```
arb
    seq
        a = 1 ; b = a
    end seq
    seq
        c = 2 ; d = c
    end seq
end arb
```

*Composition of sequential blocks (***arball***).* The following example composes ten sequences, each assigning to one element of `a` and one element of `b`.

```
arball (i = 1:10)
    seq
        a(i) = i
        b(i) = a(i)
    end seq
end arball
```

Invalid composition. The following example is not a valid **arb** composition; the two assignments are not **arb**-compatible.

```
arb
    a = 1
    b = a
end arb
```

3.10 Execution of arb-model programs

Since for **arb**-compatible program elements, their **arb** composition is semantically equivalent to their parallel composition and also to their sequential composition, programs written using sequential commands and constructors plus (valid) **arb** composition can, as noted earlier, be executed either as sequential or as parallel programs with identical results.[1] In this section we sketch how to do this in the context of practical programming languages; [30] presents additional details and examples.

Sequential execution. A program in the **arb** model can be executed sequentially; such a program can be transformed into an equivalent program in the underlying sequential notation by replacing **arb** composition with sequential composition. For Fortran 90, this is done by removing `arb` and `end arb` and transforming `arball` into nested `DO` loops.

Parallel execution. A program in the **arb** model can be executed on a shared-memory-model parallel architecture given a language construct that implements general parallel composition as defined in Definition 3.12. Language constructs consistent with this form of composition include the **par** and **parfor** constructs of CC++ [7] and the `PARALLEL DO` and `PARALLEL SECTIONS` constructs of the OpenMP proposal [33].

4 The par model and shared-memory programs

As discussed in Section 1, once we have developed a program in our **arb** model, we can transform the program into one suitable for execution on a shared-memory architecture via what we call the ***par** model*, which is based on a structured form of parallel composition with barrier synchronization that we call ***par** composition*. In our methodology, we initially write down programs using **arb** composition and sequential constructs; after applying transformations

[1] Programs that use **arb** to compose elements that are not **arb**-compatible cannot, of course, be guaranteed to have this property. As discussed in Section 3.6, we assume that the **arb** composition notation is applied only to groups of program elements that are **arb**-compatible; it is the responsibility of the programmer to ensure that this is the case.

such as those presented in Appendix C, we transform the results in **par**-model programs, which are then readily converted into programs for shared-memory architectures (by replacing **par** composition with parallel composition and our barrier synchronization construct with that provided by a selected parallel language or library). In this section we extend our model of parallel composition to include barrier synchronization, give key transformations for turning **arb**-model programs into programs using parallel composition with barrier synchronization, and briefly discuss executing such programs on shared-memory architectures.

4.1 Parallel composition with barrier synchronization

We first expand the definition of parallel composition given in Section 3 (Definition 3.12) to include barrier synchronization. Behind any synchronization mechanism is the notion of "suspending" a component of a parallel composition until some condition is met — that is, temporarily interrupting the normal flow of control in the component, and then resuming it when the condition is met. We model suspension as busy waiting, since this approach simplifies our definitions and proofs by making it unnecessary to distinguish between computations that terminate normally and computations that terminate in a deadlock situation — if suspension is modeled as a busy wait, deadlocked computations are infinite.

Specification of barrier synchronization. We first give a specification for barrier synchronization; that is, we define the expected behavior of a *barrier command* in the context of the parallel composition of programs $P_1, \ldots, P_N$. If iB_j denotes the number of times P_j has initiated the barrier command, and cB_j denotes the number of times P_j has completed the barrier command, then we require the following:

- For all j, $iB_j = cB_j$ or $iB_j = cB_j + 1$. If $iB_j = cB_j + 1$, we say that P_j is suspended at the barrier. If $iB_j = cB_j$, we say that P_j is not suspended at the barrier.
- If P_j and P_k are both suspended at the barrier, or neither P_j nor P_k is suspended at the barrier, then $iB_j = iB_k$.
- If P_j is suspended at the barrier and P_k is not suspended at the barrier, $iB_j = iB_k + 1$.
- For any n, if every P_j initiates the barrier command n times, then eventually every P_j completes the barrier command n times:

$$(\forall j :: (iB_j = cB_j + 1) \wedge (iB_j = n)) \leadsto (\forall j :: (cB_j = n)) \ .$$

We observe that this specification simply captures formally the usual meaning of barrier synchronization and is consistent with other formalizations, for example those of [2] and [36]. Most details of the specification were obtained from [38]; the overall method (in which initiations and completions of a command are considered separately) owes much to [27].

Barrier synchronization in our model. We define barrier synchronization by extending the definition of parallel composition given in Definition 3.12 and defining a new command, **barrier**. This combined definition implements a common approach to barrier synchronization based on keeping a count of processes waiting at the barrier, as in [2]. In the context of our model, we implement this approach using two protocol variables local to the parallel composition, a count Q of suspended components and a flag *Arriving* that indicates whether components are arriving at the barrier or leaving. As components arrive at the barrier, we suspend them and increment Q. When Q equals the number of components, we set *Arriving* to *false* and allow components to leave the barrier. Components leave the barrier by unsuspending and decrementing Q. When Q equals 0, we reset *Arriving* to *true*, ready for the next use of the barrier.

*Definition 4.1 (***barrier***).*

We define program **barrier** $= (V, L, InitL, A, PV, PA)$ as follows:

- $V = L \cup \{Q, Arriving\}$.
- $L = \{En, Susp\}$, where $En, Susp$ are Boolean variables.
- $InitL = (true, false)$.
- $A = \{a_{arrive}, a_{release}, a_{leave}, a_{reset}, a_{wait}\}$, where
  - a_{arrive} corresponds to a process's initiating the barrier command when fewer than $N-1$ other processes are suspended. The process should then suspend, so the action is defined by the set of state transitions $s \to s'$ such that:
    - In s, En is *true*, *Arriving* is *true*, and $Q < (N-1)$.
    - s' is s with En set to *false*, *Susp* set to *true*, and Q incremented by 1.
  - $a_{release}$ corresponds to a process's initiating the barrier command when $N-1$ other processes are suspended. The process should then complete the command and enable the other processes to complete their barrier commands as well. The action is thus defined by the set of state transitions $s \to s'$ such that:
    - In s, En is *true*, *Arriving* is *true*, and $Q = (N-1)$.
    - s' is s with En set to *false* and *Arriving* set to *false*. *Susp*, which was initially *false*, is unchanged.
  - a_{leave} corresponds to a process's completing the barrier command when at least one other process has not completed its barrier command. The action is defined by the set of state transitions $s \to s'$ such that:
    - In s, *Susp* is *true*, *Arriving* is *false*, and $Q > 1$.
    - s' is s with *Susp* set to *false* and Q decremented by 1.
  - a_{reset} corresponds to a process's completing the barrier command when all other processes have already done so. The action is defined by the set of state transitions $s \to s'$ such that:
    - In s, *Susp* is *true*, *Arriving* is *false*, and $Q = 1$.
    - s' is s with *Susp* set to *false*, *Arriving* set to *true*, and Q set to 0.

- a_{wait} corresponds to a process's busy-waiting at the barrier. The action is defined by the set of state transitions $s \to s'$ such that:
  - In s, *Susp* is *true*.
  - $s' = s$.
- $PV = \{Q, Arriving\}$.
- $PA = A$.

□

Definition 4.2 (Parallel composition with barrier synchronization).

We define parallel composition as in Section 3 (Definition 3.12), except that we add local protocol variables *Arriving* (of type Boolean) and Q (of type integer) with initial values *true* and 0 respectively.

□

Observe that this definition meets the specification given previously; a proof can be constructed by formalizing the introductory discussion preceding the definitions.

4.2 The par model

We now define a structured form of parallel composition with barrier synchronization. Previously we defined a notion of **arb**-compatibility and then defined **arb** composition as the parallel composition of **arb**-compatible components. Analogously, in this section we define a notion of **par**-compatibility and then define **par** composition as the parallel composition of **par**-compatible components. The idea behind **par**-compatibility is that the components match up with regard to their use of the barrier command — that is, they all execute the barrier command the same number of times and hence do not deadlock. Observe that while our definition is given in terms of restricted forms of the alternative (*IF*) and repetition (*DO*) constructs of Dijkstra's guarded-command language [13, 15], it applies to any programming notation with equivalent constructs.

*Definition 4.3 (**arb**-compatible, revisited).*

Programs $P_1, \ldots, P_N$ are **arb**-compatible exactly when (i) they meet the conditions for **arb**-compatibility given earlier (Definition 3.14), and (ii) for each j, P_j contains no free barriers, where program P is said to contain a *free barrier* exactly when it contains an instance of **barrier** not enclosed in a parallel composition.

□

*Definition 4.4 (**par**-compatible).*

We say programs $P_1, \ldots, P_N$ are **par**-compatible exactly when one of the following is true:

- $P_1, \ldots, P_N$ are **arb**-compatible.
- For each j,

$$P_j = Q_j; \mathbf{barrier}; R_j$$

  where $Q_1, \ldots, Q_N$ are **arb**-compatible and $R_1, \ldots, R_N$ are **par**-compatible.
- For each j,

$$P_j = \mathbf{if}\ b_j \ \rightarrow\ Q_j \ [] \ \neg b_j \ \rightarrow\ \mathit{skip}\ \mathbf{fi}$$

  where $Q_1, \ldots, Q_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.
- For each j,

$$P_j = \mathbf{if}\ b_j \ \rightarrow\ (Q_j; \mathbf{barrier}; R_j) \ [] \ \neg b_j \ \rightarrow\ \mathit{skip}\ \mathbf{fi}$$

  where $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.
- For each j,

$$P_j = \mathbf{do}\ b_j \ \rightarrow\ (Q_j; \mathbf{barrier}; R_j; \mathbf{barrier})\ \mathbf{od}$$

  where $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.

□

As with **arb**, we write $\mathbf{par}(P_1, \ldots, P_N)$ to denote the parallel composition (with barrier synchronization) of **par**-compatible elements $P_1, \ldots, P_N$. We also define a Fortran 90-compatible notation analogous to that for **arb** and a syntax **parall** analogous to **arball**.

4.3 Examples of par composition

*Composition of sequential blocks (***parall***).* The following example composes ten sequences, each assigning to one element of `a` and one element of `b`. The barrier is needed since otherwise the sequences being composed would not be **par**-compatible.

```
parall (i = 1:10)
    seq
        a(i) = i
        barrier
        b(i) = a(11-i)
    end seq
end parall
```

Invalid composition. The following example is not a valid **par** composition; the two sequences are not **par**-compatible.

```
par
    seq
        a = 1 ; barrier ; b = a
    end seq
    seq
        c = 2
    end seq
end par
```

4.4 Transforming arb-model programs into par-model programs

We now give theorems allowing us to transform programs in the **arb** model into programs in the **par** model. The versions here are suitable if the eventual goal is a program for a shared-memory architecture; versions more suitable for distributed-memory architectures are presented in Appendix B.

Theorem 4.5 (Replacement of **arb** *with* **par***).*

If $P_1, \ldots, P_N$ are **arb**-compatible,

$$\mathbf{arb}(P_1, \ldots, P_N) \sqsubseteq \mathbf{par}(P_1, \ldots, P_N)$$

□

Proof of Theorem 4.5.

Trivial.

□

Theorem 4.6 (Interchange of **par** *and sequential composition).*

If $Q_1, \ldots, Q_N$ are **arb**-compatible and $R_1, \ldots, R_N$ are **par**-compatible, then

$$\begin{array}{l} \quad \mathbf{arb}(Q_1, \ldots, Q_N); \mathbf{par}(R_1, \ldots, R_N) \\ \sqsubseteq \\ \quad \mathbf{par}(\\ \qquad (Q_1; \mathbf{barrier}; R_1), \\ \qquad \ldots, \\ \qquad (Q_N; \mathbf{barrier}; R_N) \\ \quad) \end{array}$$

□

Proof of Theorem 4.6.

First observe that both sides of the refinement have the same set of non-local variables V_{nl}. We need to show that given any maximal computation C of the right-hand side of the refinement we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. This is straightforward: In any maximal computation of the right-hand side, from the definitions of sequential composition and **barrier** we know that we can partition the computation into (1) a segment consisting of maximal computations of the Q_j's and initiations of the **barrier** command, one for each j, and (2) a segment consisting of completions of the **barrier** command, one for each j, and maximal computations of the R_j's. Segment (1) can readily be mapped to an equivalent maximal computation of $\mathbf{arb}(Q_1, \ldots, Q_N)$ by removing the barrier-initiation actions. Segment (2) can readily be mapped to an equivalent maximal computation of $\mathbf{par}(R_1, \ldots, R_N)$ by removing the first barrier-completion action for each j. We observe that this approach works even for nonterminating computations: If the right-hand side does not terminate, then either at least one Q_j does not terminate, or $\mathbf{par}(R_1, \ldots, R_N)$ does not terminate, and in either case the analogous computation of the left-hand side also does not terminate. The right-hand side cannot fail to terminate because of deadlock at the first barrier because if all the Q_j's terminate, the immediately-following executions of **barrier** terminate as well (from the specification of barrier synchronization).

□

Theorem 4.7 (Interchange of **par** *and IF, part 1).*

If $Q_1, \ldots, Q_N$ are **par**-compatible, and for all j no variable that affects b is written Q_j, then

$$
\begin{array}{ll}
 & \mathbf{if}\ b \ \rightarrow\ \mathbf{par}(Q_1, \ldots, Q_N)\ [\!]\ \neg b\ \rightarrow\ \mathit{skip}\ \mathbf{fi} \\
\sqsubseteq & \\
 & \mathbf{par}(\\
 & \quad \mathbf{if}\ b\ \rightarrow\ Q_1\ [\!]\ \neg b\ \rightarrow\ \mathit{skip}\ \mathbf{fi}, \\
 & \quad \ldots, \\
 & \quad \mathbf{if}\ b\ \rightarrow\ Q_N\ [\!]\ \neg b\ \rightarrow\ \mathit{skip}\ \mathbf{fi} \\
 &)
\end{array}
$$

□

Proof of Theorem 4.7.

Again observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. Here, such a proof uses the fact that the value of b is not changed by Q_j for any j. Since no barriers are introduced in this transformation, we do not introduce additional possibilities for deadlock.

□

Theorem 4.8 (Interchange of **par** *and IF, part 2).*

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all j no variable that affects b is written by Q_j, then

$$
\begin{array}{ll}
& \textbf{if } b \ \rightarrow \ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N)) \ [] \ \neg b \ \rightarrow \ \textit{skip} \ \textbf{fi} \\
\sqsubseteq & \\
& \textbf{par}(\\
& \quad \textbf{if } b \ \rightarrow \ (Q_1; \textbf{barrier}; R_1) \ [] \ \neg b \ \rightarrow \ \textit{skip} \ \textbf{fi}, \\
& \quad \ldots, \\
& \quad \textbf{if } b \ \rightarrow \ (Q_N; \textbf{barrier}; R_N) \ [] \ \neg b \ \rightarrow \ \textit{skip} \ \textbf{fi} \\
&)
\end{array}
$$

□

Proof of Theorem 4.8.

Again observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. The barrier introduced in the transformation cannot deadlock for reasons similar to those for the transformation of Theorem 4.6.

□

Theorem 4.9 (Interchange of **par** *and DO).*

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all j no variable that affects b is written by Q_j, then

$$
\begin{array}{l}
\quad \mathbf{do}\ b \ \rightarrow\ (\mathbf{arb}(Q_1, \ldots, Q_N); \mathbf{par}(R_1, \ldots, R_N))\ \mathbf{od} \\
\sqsubseteq \\
\quad \mathbf{par}(\\
\qquad \mathbf{do}\ b \ \rightarrow\ (Q_1; \mathbf{barrier}; R_1; \mathbf{barrier})\ \mathbf{od}, \\
\qquad \ldots, \\
\qquad \mathbf{do}\ b \ \rightarrow\ (Q_N; \mathbf{barrier}; R_N; \mathbf{barrier})\ \mathbf{od} \\
\quad)
\end{array}
$$

□

Proof of Theorem 4.9.

First observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. The proof makes use of the restrictions on when variables that affect b can be written. For terminating computations, the proof can be constructed using the standard unrolling of the repetition command (as in [18] or [15]) together with Theorem 4.6 and Theorem 4.8. For nonterminating computations, the proof must consider two classes of computations: those that fail to terminate because an iteration of one of the loops fails to terminate, and those that fail to terminate because one of the loops iterates forever. In both cases, however, the computation can be mapped onto an infinite (and therefore, in our model, equivalent) computation of the left-hand side.

□

Example of applying transformations. Let P be the following program:

```
do while (x < 100)
    arb
        a = a * 2
        b = b + 1
    end arb
    par
        x = max(a, b)
        skip
    end par
end do
```

Then P is refined by the following:

```
par
    do while (x < 100)
        a = a * 2 ; barrier ; x = max(a, b) ; barrier
    end do
    do while (x < 100)
        b = b + 1 ; barrier ; skip ; barrier
    end do
end par
```

Additional examples of applying these transformations are given in [30].

4.5 Executing par-model programs

It is clear that **par** composition as described in this section is implemented by general parallel composition (as described in Section 3.10) plus a barrier synchronization that meets the specification of Section 4.1. Thus, we can transform a program in the **par** model into an equivalent program in any language with constructs that implement composition and barrier synchronization in a way consistent with our definitions (which in turn are consistent with the usual meaning of parallel composition with barrier synchronization). Examples of such constructs are the `PARALLEL DO`, `PARALLEL SECTIONS`, and `BARRIER` constructs of the OpenMP proposal [33]. Examples of conversions are given in [30].

5 The subset par model and distributed-memory programs

As discussed in Section 1, once we have developed a program in our **arb** model, we can transform the program into one suitable for execution on a distributed-memory–message-passing architecture via what we call the *subset **par** model*, which is a restricted form of the **par** model discussed in Section 4. In our methodology, we apply a succession of transformations to an **arb**-model program to produce a program in the subset **par** model and then transform the result into a program for a distributed-memory–message-passing architecture. In this section we extend our model of parallel composition to include message-passing operations, define a restricted subset of the **par** model that corresponds more directly to distributed-memory architectures, discuss transforming programs in the resulting subset **par** model into programs using parallel composition with message-passing, and briefly discuss executing such programs on distributed-memory–message-passing architectures.

5.1 Parallel composition with message-passing

We first expand the definition of parallel composition given in Section 3 to include message-passing.

Specification of message-passing. We define message-passing for $P_1, \ldots, P_N$ composed in parallel in a way compatible with single-sender–single-receiver channels with infinite slack (i.e., infinite capacity). Every message operation (send or receive) specifies a sender and a receiver, and while a receive operation suspends if there is no message to receive, a send operation never suspends. Messages are received in the order in which they are sent and are not received before they are sent. That is, if we let $nS_{j,k}$ denote the number of send operations from P_j to P_k performed, $iR_{j,k}$ denote the number of receive operations from P_j to P_k initiated, and $cR_{j,k}$ denote the number of such receive operations completed, then we can write the desired specification as follows:

- $iR_{j,k} = cR_{j,k}$ or $iR_{j,k} = cR_{j,k} + 1$ for all j, k.
- Messages are not received before they are sent: $nS_{j,k} \geq cR_{j,k}$ for all j, k.
- Messages are received in the order in which they are sent: The n-th message received by P_j from P_k is identical with the n-th message sent from P_k to P_j.
- If n messages are sent from P_k to P_j, and P_j initiates n receive operations for messages from P_k, then all will complete:

$$(nS_{j,k} \geq n) \wedge (iR_{j,k} = n)) \rightsquigarrow (cR_{j,k} = n) \enspace .$$

We observe that this specification, like the one for barrier synchronization in Section 4, simply captures formally the usual meaning of this type of message passing, and is consistent with other formalizations, for example those of [2] and [35]. The terminology ("slack") and overall method (in which initiations and completions of a command are considered separately) are based on [27].

Message-passing in our model. Like many other implementations of message-passing, for example that of [2], our definition represents channels as queues:

We define for each ordered pair (P_j, P_k) a queue $C_{j,k}$ whose elements represent messages in transit from P_j to P_k. Message sends are then represented as enqueue operations and message receives as (possibly suspending) dequeue operations. Elements of $C_{j,k}$ take the form of pairs $(\mathit{Type}, \mathit{Value})$. Just as we did in Section 4, we model suspension as busy waiting.

*Definition 5.1 (***send***).*

We define program **send** $= (V, L, \mathit{InitL}, A, PV, PA)$ as follows:

- $V = L \cup \{\mathit{OutP}_1, \ldots, \mathit{OutP}_N, \mathit{Rcvr}, \mathit{Type}, \mathit{Value}\}$, where each OutP_j ("outport j") is a variable of type queue, Rcvr is an integer variable, Type is a type, and Value is a variable of type Type. Variables $\mathit{OutP}_1, \ldots, \mathit{OutP}_N$ are to be shared with the enclosing parallel composition, as described later, while variables Rcvr, Type, Value are to be shared with the enclosing sequential composition. (I.e., it is assumed that **send** is composed in sequence with assignment statements that assign appropriate values to Rcvr, Type, and Value.)

- $L = \{En\}$, where En is a Boolean variable.
- $InitL = (true)$.
- $A = \{a_{snd}\}$, where
  - a_{snd} corresponds to a process's sending a message $(Type, Value)$ to process P_{Rcvr}. The action is defined by the set of state transitions $s \to s'$ such that:
    * In s, En is $true$.
    * s' is s with En set to $false$ and $(Type, Value)$ enqueued (appended) to $OutP_{Rcvr}$.
- $PV = \{OutP_1, \ldots, OutP_N\}$.
- $PA = A$.

□

*Definition 5.2 (**recv**).*

We define program **recv** $= (V, L, InitL, A, PV, PA)$ as follows:

- $V = L \cup \{InP_1, \ldots, InP_N, Sndr, Type, Value\}$, where each InP_j ("inport j") is a variable of type queue, $Sndr$ is an integer variable, $Type$ is a type, and $Value$ is a variable of type $Type$. Variables $InP_1, \ldots, InP_N$ are to be shared with the enclosing parallel composition, as described later, while variables $Sndr$, $Type$, $Value$ are to be shared with the enclosing sequential composition, similarly to the analogous variables of **send**.
- $L = \{En\}$, where En is a Boolean variable.
- $InitL = (true)$.
- $A = \{a_{rcv}, a_{wait}\}$, where
  - a_{rcv} corresponds to a process's receiving a message $(Type, Value)$ from process P_{Sndr}. The action is defined by the set of state transitions $s \to s'$ such that:
    * In s, En is $true$ and InP_{Sndr} is not empty.
    * s' is s with En set to $false$ and $(Type, Value)$ and InP_{Sndr} set to the values resulting from dequeueing an element from InP_{Sndr}.
  - a_{wait} corresponds to a process's waiting for a message from process P_{Sndr}. The action is defined by the set of state transitions $s \to s'$ such that:
    * In s, En is $true$ and InP_{Sndr} is empty.
    * $s' = s$.
- $PV = \{InP_1, \ldots, InP_N\}$.
- $PA = A$.

□

Definition 5.3 (Parallel composition with message-passing).

We define parallel composition as in Section 3 (Definition 3.12), except that we add local protocol variables $C_{j,k}$ (of type queue), one for each ordered pair (P_j, P_k), with initial values of "empty", and we perform the following additional modifications on the component programs P_j:

– We replace variables $OutP_1, \ldots, OutP_N$ in V_j with $C_{j,1}, \ldots, C_{j,N}$, and we make the same replacement in actions a derived from a_{snd}.
– We replace variables $InP_1, \ldots, InP_N$ in V_j with $C_{1,j}, \ldots, C_{N,j}$, and we make the same replacement in actions a derived from a_{rcv} and a_{wait}.

□

Observe that this definition clearly meets the specification given earlier.

5.2 The subset par model

We define the subset **par** model such that a computation of a program in this model may be thought of as consisting of an alternating sequence of (i) blocks of computation in which each component operates independently on its local data, and (ii) blocks of computation in which values are copied between components, separated by barrier synchronization, as illustrated by Figure 3. We refer to a

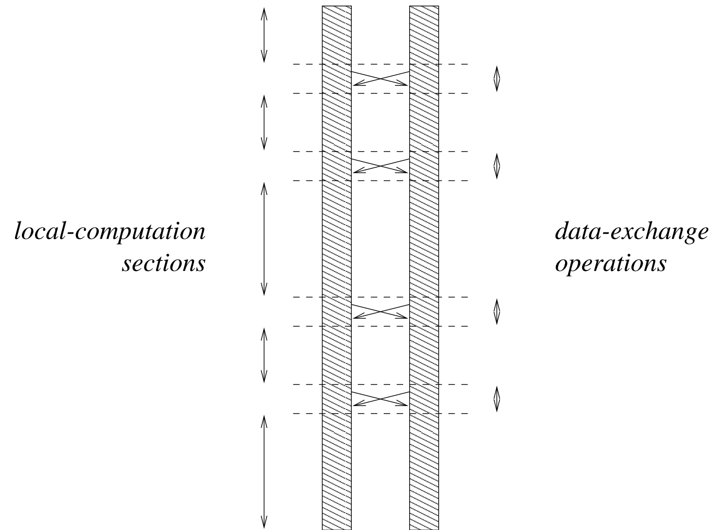

Fig. 3. A computation of a subset-**par**-model program. Shaded vertical bars represent computations of processes, arrows represent copying of data between processes, and dashed horizontal lines represent barrier synchronization.

block of the first variety as a *local-computation section* and to a block of the second variety (together with the preceding and succeeding barrier synchronizations) as a *data-exchange operation*.

That is, a program in the subset **par** model is a composition $\mathbf{par}(P_1, \ldots, P_N)$, where $P_1, \ldots, P_N$ are subset-**par**-compatible as defined by the following.

Definition 5.4 (Subset **par***-compatibility).*

$P_1, \ldots, P_N$ are subset-**par**-compatible exactly when (i) $P_1, \ldots, P_N$ are **par**-compatible, (ii) the variables V of the composition (excluding the protocol variables representing message channels) are partitioned into disjoint subsets $W_1, \ldots, W_N$, and (iii) exactly one of the following holds:

- $P_1, \ldots, P_N$ are **arb**-compatible and each P_j reads and writes only variables in W_j.
- For each j,

$$P_j = Q_j; \mathbf{barrier}; Q'_j; \mathbf{barrier}; R_j$$

  where
  - $Q_1, \ldots, Q_N$ are **arb**-compatible.
  - Each Q_j reads and writes only variables in W_j.
  - Each Q'_j is an **arb**-compatible set of assignment statements $x_k := x_j$ such that x_j is an element of W_j and x_k is an element of W_k for some k (possibly $k = j$).
  - $R_1, \ldots, R_N$ are subset-**par**-compatible.
- For each j, $b_j \in W_j$ and

$$P_j = \mathbf{if}\ b_j \ \rightarrow\ Q_j \ [] \ \neg b_j \ \rightarrow\ \mathit{skip}\ \mathbf{fi}$$

  where $Q_1, \ldots, Q_N$ are subset-**par**-compatible.
- For each j, $b_j \in W_j$ and

$$P_j = \mathbf{do}\ b_j \ \rightarrow\ Q_j\ \mathbf{od}$$

  where $Q_1, \ldots, Q_N$ are subset-**par**-compatible.

□

5.3 Example of subset par composition

The following example computes the maximum of four elements using recursive doubling:

```
integer a(4), part(2), part_copy(2), m(2)
arb
    part(1) = max(a(1), a(2))
    part(2) = max(a(3), a(4))
end arb
arb
    part_copy(1) = part(2)
    part_copy(2) = part(1)
end arb
arb
    m(1) = max(part(1), part_copy(1))
    m(2) = max(part_copy(2), part(2))
end arb
```

5.4 Transforming subset-par-model programs into programs with message-passing

We can transform a program in the subset **par** model into a program for a distributed-memory–message-passing architecture by mapping each component P_j onto a process j and making the following additional changes:

- Map each element W_j of the partition of V to the address space for process j.
- Convert each data-exchange operation (consisting of a set of (**barrier**; Q'_j; **barrier**) sequences, one for each component P_j) into a collection of message-passing operations, in which each assignment $x_j := x_k$ is transformed into a pair of message-passing commands: a **send** command in k specifying $Rcvr = j$, and a **recv** command in j specifying $Sndr = k$.
- Optionally, for any pair (P_j, P_k) of processes, concatenate all the messages sent from P_j to P_k as part of a data-exchange operation into a single message, replacing the collection of (send, receive) pairs from P_j to P_k with a single (send, receive) pair.

Such a program refines the original program: Each send–receive pair of operations produces the same result as the assignment statement from which it was derived (as discussed in [21] and [28]), and the **arb**-compatibility of the assignments ensures that these pairs can be executed in any order without changing the result. Replacing barrier synchronization with the weaker pairwise synchronization implied by these pairs of message-passing operations also preserves program correctness; we can construct a proof of this claim by using the techniques of Section 3 and our definitions of barrier synchronization and message-passing. A similar theorem and its proof are given in [29].

Example. If P is the recursive-doubling example program of Section 5.3, P is refined by the following subset-**par**-model program P' with variables partitioned into

- $W_1 = \{\texttt{a}(1:2), \texttt{part}(1), \texttt{part_copy}(1), \texttt{m}(1)\}$ and
- $W_2 = \{\texttt{a}(3:4), \texttt{part}(2), \texttt{part_copy}(2), \texttt{m}(2)\}$:

```
arb
    seq
        part(1) = max(a(1), a(2))
        barrier ; part_copy(1) = part(2) ; barrier
        m(1) = max(part(1), part_copy(1))
    end seq
    seq
        part(2) = max(a(3), a(4))
        barrier ; part_copy(2) = part(1) ; barrier
        m(2) = max(part_copy(2), part(2))
    end seq
end arb
```

which is in turn refined by the following message-passing program P'':

```
arb
    seq
        part(1) = max(a(1), a(2))
        send ("integer", part(1)) to (P2)
        recv (type, part_copy(1)) from (P2)
        m(1) = max(part(1), part_copy(1))
    end seq
    seq
        part(2) = max(a(3), a(4))
        send ("integer", part(2)) to (P1)
        recv (type, part_copy(2)) from (P1)
        m(2) = max(part(2), part_copy(2))
    end seq
end arb
```

5.5 Executing subset-par-model programs

We can use the transformation of the preceding section to transform programs in the subset **par** model into programs in any language that supports multiple-address-space parallel composition with single-sender–single-receiver message-passing. Examples include Fortran M [16] (which supports multiple-address-space parallel composition via process blocks and single-sender–single-receiver message-passing via channels) and MPI [31] (which assumes execution in an environment of multiple-address-space parallel composition and supports single-sender–single-receiver message-passing via tagged point-to-point sends and receives).

Example. Program P'' from Section 5.4 can be implemented by the following Fortran M program:

```
program main
    integer a(4)
    inport (integer) inp(2)
    outport (integer) outp(2)
    channel (outp(1), inp(2))
    channel (outp(2), inp(1))
    processes
        process call P(a(1:2), inp(1), outp(1))
        process call P(a(3:4), inp(2), outp(2))
    end processes
end

process P(a, inp, outp)
    integer a(2)
```

```
        inport (integer) inp
        outport (integer) outp
        integer part, part_copy, m
        part = max(a(1), a(2))
        send (outp) part
        receive (inp) part_copy
        m = max(part, part_copy)
    end process
```

6 Related work

Program development via stepwise refinement. Other researchers, for example Back [3, 4] and Martin [28], have addressed stepwise refinement for parallel programs. Our work is somewhat simpler than many approaches because we deal only with specifications that can be stated in terms of initial and final states, rather than also addressing ongoing program behavior (e.g., safety and progress properties).

Operational models. Our operational model is based on defining programs as state-transition systems, as in the work of Chandy and Misra [9], Lynch and Tuttle [24], Lamport [23], Manna and Pnueli [26], and Pnueli [34]. Our model is designed to be as simple as possible while retaining enough generality to support all aspects of our programming model.

Parallel programming models. Programming models similar in spirit to ours have been proposed by Valiant [39] and Thornley [37]; our model differs in that we provide a more explicit supporting theoretical framework and in the use we make of archetypes.

Automatic parallelization of sequential programs. Our work is in many respects complementary to efforts to develop parallelizing compilers, for example Fortran D [12]. The focus of such work is on the automatic detection of exploitable parallelism, while our work addresses how to exploit parallelism once it is known to exist. Our theoretical framework could be used to prove not only manually-applied transformations but also those applied by parallelizing compilers.

Programming skeletons and patterns. Our work is also in some respects complementary to work exploring the use of programming skeletons and patterns in parallel computing, for example that of Cole [11] and Brinch Hansen [6]. We also make use of abstractions that capture exploitable commonalities among programs, but we use these abstractions to guide a program development methodology based on program transformations.

7 Conclusions

We believe that our operational model, presented in Section 3, forms a suitable framework for reasoning about program correctness and transformations, particularly transformations between our different programming models. Proofs of the theorems of Section 3, sketched here and presented in detail in [30], demonstrate that this model can be used as the basis for rigorous and detailed proofs. Our programming model, which is based on identifying groups of program elements whose sequential composition and parallel composition are semantically equivalent, together with the collection of transformations presented in [30] for converting programs in this model to programs for typical parallel architectures, provides a framework for program development that permits much of the work to be done with well-understood and familiar sequential tools and techniques. A discussion of how our approach can simplify the task of producing correct parallel applications is outside the scope of this paper, but [30] presents examples of its use in developing example and real-world applications with good results.

Much more could be done, particularly in exploring providing automated support for the transformations we describe and in identifying additional useful transformations, but the results so far are encouraging, and we believe that the work as a whole constitutes an effective unified theory/practice framework for parallel application development.

Acknowledgments

Thanks go to Mani Chandy for his guidance and support of the work on which this paper is based, and Eric Van de Velde for the book [40] that was an early inspiration for this work.

References

1. J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference.* Intertext Publications : McGraw-Hill Book Company, 1992.
2. G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.
3. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.
4. R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.
5. R. Bagrodia, K. M. Chandy, and M. Dhagat. UC — a set-based language for data-parallel programming. *Journal of Parallel and Distributed Computing*, 28(2):186–201, 1995.

6. P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.
7. K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
8. K. M. Chandy and B. L. Massingill. Parallel program archetypes. Technical Report CS-TR-96-28, California Institute of Technology, 1997.
9. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
10. A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.
11. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
12. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 82(2):244–263, 1993.
13. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivations of programs. *Communications of the ACM*, 18(8):453–457, 1975.
14. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.
15. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
16. I. T. Foster and K. M. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
18. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
19. High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1–2):1–170, 1993.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
21. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
22. International Standards Organization. ISO/IEC 1539:1991 (E), Fortran 90, 1991.
23. L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
24. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987.
25. B. J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
26. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.
27. A. J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.
28. A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
29. B. L. Massingill. Experiments with program parallelization using archetypes and stepwise refinement. Technical Report UF-CISE-98-012, University of Florida, 1998.

30. B. L. Massingill. A structured approach to parallel programming. Technical Report CS-TR-98-04, California Institute of Technology, 1998. Ph.D. thesis. Available as ⟨ `ftp://ftp.cs.caltech.edu/tr/cs-tr-98-04.ps.Z` ⟩.
31. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.
32. J. M. Morris. Piecewise data refinement. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*. Addison-Wesley Publishing Company, Inc., 1990.
33. OpenMP Partners. The OpenMP standard for shared-memory parallel directives, 1998. ⟨ `http://www.openmp.org` ⟩.
34. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
35. P. A. G. Sivilotti. A verified integration of imperative parallel programming paradigms in an object-oriented language. Technical Report CS-TR-93-21, California Institute of Technology, 1993.
36. P. A. G. Sivilotti. Reliable synchronization primitives for Java threads. Technical Report CS-TR-96-11, California Institute of Technology, 1996.
37. J. Thornley. A parallel programming model with sequential semantics. Technical Report CS-TR-96-12, California Institute of Technology, 1996.
38. J. Thornley and K. M. Chandy. Barriers: Specification. Unpublished document.
39. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
40. E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.

A Some commands of Dijkstra's guarded-command language in our model

This section sketches definitions in our model for some of the commands of Dijkstra's guarded-command language [13, 15]. Definitions of additional constructors appear in [30].

Definition A.1 (Skip).

We define program $skip = (V, L, InitL, A, PV, PA)$ as follows:

- $V = L$.
- $L = \{En_{skip}\}$, where En_{skip} is a Boolean variable.
- $InitL = (true)$.
- $A = \{a\}$, where

$$\begin{aligned} I_a &= \{En_{skip}\} \\ O_a &= \{En_{skip}\} \\ R_a &= \{((true), (false))\} \end{aligned}$$

- $PV = \{\}$.
- $PA = \{\}$.

□

Definition A.2 (Assignment).

We define program $P = (V, L, InitL, A, PV, PA)$ for $(y := E)$ as follows:

- $V = \{v_1, \ldots, v_N\} \cup \{y\} \cup L$, where $\{v_1, \ldots, v_N\} = \{v : affects.(v, E) : v\}$.
- $L = \{En_P\}$, where En_P is a Boolean variable not otherwise occurring in V.
- $InitL = (true)$.
- $A = \{a\}$, where

$$\begin{aligned} I_a &= \{En_P\} \cup \{v_1, \ldots, v_N\} \\ O_a &= \{En_P, y\} \\ R_a &= \{x_1, \ldots, x_N :: ((true, x_1, \ldots, x_N), (false, E.(x_1, \ldots, x_N))\} \end{aligned}$$

  and $x_1, \ldots, x_N$ is an assignment of values to the variables in $v_1, \ldots, v_N$.
- $PV = \{\}$.
- $PA = \{\}$.

□

Definition A.3 (Abort).

We define program $abort = (V, L, InitL, A, PV, PA)$ as follows:

- $V = L$
- $L = \{En_{abort}\}$, where En_{abort} is a Boolean variable.
- $InitL = (true)$.
- $A = \{a\}$, where

$$\begin{aligned} I_a &= \{En_{abort}\} \\ O_a &= \{\} \\ R_a &= \{(true), ()\} \end{aligned}$$

- $PV = \{\}$.
- $PA = \{\}$.

□

B More about transforming arb-model programs into par-model programs

This section presents versions of some of the theorems of in Section 4.4 more suitable for transforming programs for distributed-memory architectures. For proofs of theorems refer to [30].

Lemma B.1 (Interchange of **par** *and IF, part 1, with duplicated variables).*

If $Q_1, \ldots, Q_N$ and b are as for Theorem 4.7, and $b_1, \ldots, b_N$ are Boolean expressions such that for $j \neq k$ no variable that affects b_j is written by Q_k, then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

$$\begin{array}{l} \quad \textbf{if } b \;\rightarrow\; \textbf{par}(Q_1, \ldots, Q_N) \;[]\; \neg b \;\rightarrow\; \mathit{skip} \;\textbf{fi} \\ \sqsubseteq \\ \quad \textbf{par}(\\ \quad\quad \textbf{if } b_1 \;\rightarrow\; Q_1 \;[]\; \neg b_1 \;\rightarrow\; \mathit{skip} \;\textbf{fi}, \\ \quad\quad \ldots, \\ \quad\quad \textbf{if } b_N \;\rightarrow\; Q_N \;[]\; \neg b_N \;\rightarrow\; \mathit{skip} \;\textbf{fi} \\ \quad) \end{array}$$

□

Proof of Lemma B.1.

This lemma follows from Theorem 4.7 and exploitation of copy consistency as discussed in Section C.2.

□

Lemma B.2 (Interchange of **par** *and IF, part 2, with duplicated variables).*

If $Q_1, \ldots, Q_N$, $R_1, \ldots, R_N$, and b are as for Theorem 4.8, and $b_1, \ldots, b_N$ are Boolean expressions such that for $j \neq k$ no variable that affects b_j is written by Q_k, then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

$$
\begin{array}{ll}
 & \textbf{if}\ b\ \rightarrow\ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N))\ [\!]\ \neg b\ \rightarrow\ \textit{skip}\ \textbf{fi} \\
\sqsubseteq & \\
 & \textbf{par}(\\
 & \quad \textbf{if}\ b_1\ \rightarrow\ (Q_1; \textbf{barrier}; R_1)\ [\!]\ \neg b_1\ \rightarrow\ \textit{skip}\ \textbf{fi}, \\
 & \quad \ldots, \\
 & \quad \textbf{if}\ b_N\ \rightarrow\ (Q_N; \textbf{barrier}; R_N)\ [\!]\ \neg b_N\ \rightarrow\ \textit{skip}\ \textbf{fi} \\
 &)
\end{array}
$$

□

Proof of Lemma B.2.

Analogous to Lemma B.1.

□

Lemma B.3 (Interchange of **par** *and DO, with duplicated variables).*

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all $k \neq j$ no variable that affects b_j is written by Q_k, and $(\forall j :: (b_j = b))$ is an invariant of the loop

$$\textbf{do}\ b\ \rightarrow\ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N))\ \textbf{od}$$

then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

$$
\begin{array}{ll}
 & \textbf{do}\ b\ \rightarrow\ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N))\ \textbf{od} \\
\sqsubseteq & \\
 & \textbf{par}(\\
 & \quad \textbf{do}\ b_1\ \rightarrow\ (Q_1; \textbf{barrier}; R_1, \textbf{barrier})\ \textbf{od}, \\
 & \quad \ldots, \\
 & \quad \textbf{do}\ b_N\ \rightarrow\ (Q_N; \textbf{barrier}; R_N, \textbf{barrier})\ \textbf{od} \\
 &)
\end{array}
$$

□

Proof of Lemma B.3.

Analogous to Lemma B.1.

□

Example of applying transformations. Let P be the following program:

```
x = max(a, b)
do while (x < 100)
    arb
        a = a * 2
        b = b + 1
    end arb
    par
        x = max(a, b)
        skip
    end par
end do
```

Then P is refined (using the data-duplication techniques of Section C.2) by the following:

```
arb
    x1 = max(a, b)
    x2 = max(a, b)
end arb
do while (x1 < 100)
    arb
        a = a * 2
        b = b + 1
    end arb
    par
        x1 = max(a, b)
        x2 = max(a, b)
    end par
end do
```

which in turn is refined (using Theorem B.3) by the following:

```
arb
    x1 = max(a, b)
    x2 = max(a, b)
end arb
par
    do while (x1 < 100)
        a = a * 2 ; barrier ; x1 = max(a, b) ; barrier
    end do
    do while (x2 < 100)
        b = b + 1 ; barrier ; x2 = max(a, b) ; barrier
    end do
end par
```

which again in turn is refined by the following:

```
par
    seq
        x1 = max(a, b)
        barrier
        do while (x1 < 100)
            a = a * 2 ; barrier ; x1 = max(a, b) ; barrier
        end do
    end seq
    seq
        x2 = max(a, b)
        barrier
        do while (x2 < 100)
            b = b + 1 ; barrier ; x2 = max(a, b) ; barrier
        end do
    end seq
end par
```

C Some example transformations

Section 2.5 sketches our program-development strategy. A key element of that strategy, and one not discussed in detail in this paper, is the sequence of transformations that convert the original **arb**-model program into one that can be transformed into a program in the **par** or subset **par** model and thence into a program for the target architecture. A collection of transformations useful in this process appears in [30]; we summarize a few here.

C.1 Change of granularity

If the number of elements in an **arb** composition is large compared to the number of processors available for execution, and the cost of creating a separate thread for each element of the composition is relatively high, then we can improve the efficiency of the program by reducing the number of threads required, that is, by changing the granularity of the program.

We can change the granularity of an **arb**-model program by transforming an **arb** composition of N elements into a combination of **arb** composition (of fewer than N elements) and sequential composition, as described in the following theorem.

Theorem C.1 (Change of granularity).

If $P_1, \ldots, P_N$ are **arb**-compatible, and we have integers $j_1, j_2, \ldots, j_M$ such that $(1 < j_1) \wedge (j_1 < j_2) \wedge \ldots \wedge (j_M < N)$, then

$$
\begin{array}{ll}
& \mathbf{arb}(P_1, \ldots, P_N) \\
\sim & \\
& \mathbf{arb}(\\
& \quad \mathbf{seq}(P_1, \ldots, P_{j_1}), \\
& \quad \mathbf{seq}(P_{j_1+1}, \ldots, P_{j_2}), \\
& \quad \ldots, \\
& \quad \mathbf{seq}(P_{j_M+1}, \ldots, P_N) \\
&)
\end{array}
$$

□

Proof of Theorem C.1.

See [30].

□

C.2 Data distribution and duplication

In order to transform a program in the **arb** model into a program suitable for execution on a distributed-memory architecture, we must partition its variables into distinct groups, each corresponding to an address space (and hence to a process). Section 5 describes the characteristics such a partitioning should have in order to permit execution on a distributed-memory architecture; in this section we discuss only the mechanics of the partitioning, that is, transformations that effect partitioning while preserving program correctness. These transformations fall into two categories: data distribution, in which variables of the original program are mapped one-to-one onto variables of the transformed program; and data duplication, in which the map is one-to-many, that is, in which some variables of the original program are duplicated in the transformed program.

Data distribution. The transformations required to effect data distribution are in essence renamings of program variables, in which variables of the original program are mapped one-to-one to variables of the transformed program. The most typical use of data distribution is in partitioning non-atomic data objects such as arrays: Each array is divided into *local sections*, one for each process, and a one-to-one map is defined between the elements of the original array and the elements of the (disjoint) union of the local sections. That such a renaming operation does not change the meaning of the program is clear, although if elements of the array are referenced via index variables, some care must be taken to ensure that they (the index variables) are transformed in a way consistent with the renaming/mapping.

Data duplication. The transformations involved in data duplication are less obviously semantics-preserving than those involved in data distribution. The goal of such a transformation is to replace a single variable with multiple copies, such that "copy consistency is maintained when it matters." We use the term *(re-)establishing copy consistency* to refer to (re-)establishing the property that all of the copies have the same value (and that their value is the same as that of the original variable at an analogous point in the computation). In the transformed program, all copies have the same initial value as the initial value of the original variable (thereby establishing copy consistency), and any reference to a copy that changes its value is followed by program actions to assign the new value to the other copies as well (thereby re-establishing copy consistency when it is violated). Whenever copy consistency holds, a read reference to the original variable can be transformed into a read reference to any one of the copies without changing the meaning of the program.

We can accomplish such a transformation using the techniques of data refinement, as described in [32]. We begin with the following data-refinement transformation: Given program P with local variables L, duplicating variable w in L means producing a program P' with variables

$$L' = L \setminus \{w\} \cup \{w^{(1)}, \ldots, w^{(N)}\}$$

(where N is the number of copies desired and $w^{(1)}, \ldots, w^{(N)}$ are the copies of w), such that $P \sqsubseteq P'$. It is simplest to think in terms of renaming w to $w^{(1)}$ and then introducing variables $w^{(2)}, \ldots, w^{(N)}$; it is then clear what it means for P' (with variable $w^{(1)}$) to meet the same specification as P (with variable w).

Using the techniques of data refinement, we can produce such a program P' by defining the abstraction invariant

$$\forall j : 2 \leq j \leq N : w^{(j)} = w^{(1)}$$

and transforming P as follows:

- Assign the same initial value to each copy $w^{(j)}$ in $InitL'$ that was assigned to w in $InitL$, and replace any assignment $w := E$ in P with the multiple assignment
$$w^{(1)}, \ldots, w^{(N)} := E^{(1)}, \ldots, E^{(N)}$$
where $E^{(k)} = E[w/w^{(j)}]$ (j is arbitrary and can be different for different values of k). Observe that multiple assignment can be implemented as a sequence of assignments, possibly using temporary variables if w affects E.
- Replace any other reference to w in P with a reference to $w^{(j)}$, where j is arbitrary.

The first replacement rule ensures that the abstraction invariant holds after each command; the second rule makes use of the invariant. In our informal terminology, the abstraction invariant states that copy consistency holds, and the two replacement rules respectively (re-)establish and exploit copy consistency.

Let P' be the result of applying these refinement rules to P. Then $P \sqsubseteq P'$. We do not give a detailed proof, but such a proof could be produced using the rules of data refinement (as given in [32]) and structural induction on P.

For our purposes, however, P' as just defined may not be quite what we want, since in some situations it would be advantageous to postpone re-establishing copy consistency (e.g., if there are several duplicated variables, it might be advantageous to defer re-establishing copy consistency until all have been assigned new values), if we can do so without losing the property that $P \sqsubseteq P'$. We observe, then, that

$$\begin{array}{ll} & (w^{(1)}, \ldots, w^{(N)} := E^{(1)}, \ldots, E^{(N)}) \; ; \; Q \\ \sqsubseteq & \\ & w^{(k)} := E^{(k)} \; ; \; Q \; ; \; (w^{(1)}, \ldots, w^{(k-1)}, w^{(k+1)}, \ldots, w^{(N)} \; := \; w^{(k)}, \ldots, w^{(k)}) \end{array}$$

as long as for all $j \neq k$, $w^{(j)}$ is not among the variables read or written by Q. The argument for the correctness of this claim is similar to that used to prove Theorem 3.21 in Section 3.7.

We can thus give the following replacement rules for duplicating variable w in an **arb**-model program:

- Replace $w := E$ with

$$\mathbf{arb}(w^{(1)} \; := \; E[w/w^{(1)}] \; , \; \ldots \; , \; w^{(N)} := E[w/w^{(N)}]) \; .$$

- If w is not written by any of $P_1, \ldots, P_N$, replace $\mathbf{arb}(P_1, \ldots, P_N)$ with

$$\mathbf{arb}(P_1[w/w^{(1)}], \ldots, P_N[w/w^{(N)}]) \; .$$

- If w is written by P_k but neither read nor written by any other P_k, replace $\mathbf{arb}(P_1, \ldots, P_N)$ with

$$\begin{array}{l} \mathbf{arb}(P_1, \ldots, P_k[w/w^{(k)}], \ldots, P_N) \; ; \\ \mathbf{arb}(w^{(1)} := w^{(k)}, \; \ldots, \; w^{(k-1)} := w^{(k)}, \; w^{(k+1)} := w^{(k)}, \; \ldots, \; w^{(N)} := w^{(k)}) \; . \end{array}$$