A Structured Approach to Parallel Programming: Methodology and Models*

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 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

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

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.

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.

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 in [8] or [10], e.g.) 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 [3]) 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.

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.

Supporting framework for proving transformations correct. Some of the transformations mentioned in the preceding section — those within the **arb** model — can be proved correct using the techniques of sequential stepwise refinement (as defined in [10] or [11], e.g.). 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.

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 [9] 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 details of the parallel programs. Archetypes are described in more detail in [15].

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 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 [15]. 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. This section first presents our operational model for parallel programs and then discusses **arb**-compatibility and **arb** composition. In this paper we present this material with a minimum of mathematical notation and no proofs; a more formal treatment of the material, including proofs, appears in [15]. The notation we use for definitions and theorems for our programming models is based on that of Dijkstra's guarded-command language [8].

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. Treating programs as state-transition systems is not a new approach; it has been used by many other researchers (as mentioned in Section 6) 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 obviously helpful in reasoning about parallel programs, but they are applicable to sequential programs as well. Our operational model builds on this basic view of program execution, presented in a way specifically aimed at facilitating the stating and proving of our main theorems.

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, and 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). (I.e., if v is in PV, and $a = (I_a, O_a, R_a)$ is an action such that $v \in O_a$, a must be in PA.) 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. Variables in PV can include both local and non-local variables.
- $PA \subseteq A$ are *protocol actions*. Protocol actions may modify both protocol and 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 $s \xrightarrow{a} s'$, as follows: $s \xrightarrow{a} s'$ 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. □

Examples of defining the commands and constructs of a particular programming notation in terms of our model are presented in [15].

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 $s \xrightarrow{a} s'$. □

Definition 3.4 (Computation). If $P = (V, L, InitL, A, PV, PA)$, 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, $s_{j-1} \xrightarrow{a_j} s_j$. 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 (in which case we say C is finite) or ∞ (in which case we say C is infinite).

– 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 $(E = k)$. □

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 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. 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. 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 $P \sim P'$. The following definition and theorem provide a sufficient condition for showing that $P_1 \sqsubseteq P_2$ in our semantics.

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: (i) 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; and (ii) either (a) both C_1 and C_2 are infinite, or (b) 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. □

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 $(V_1 \setminus L_1)$. □

Program composition. Before giving definitions of sequential and parallel composition in terms of our model, we need the following definition formalizing conditions under which it makes sense to compose programs.

Definition 3.10 (Composability of programs). We say that a set of programs $P_1, \ldots, P_N$ *can be composed* exactly when (i) 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), (ii) any action that appears in more than one program is defined in the same way in all the programs in which it appears, and (iii) 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.
- $InitL$ 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: (i) 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. (ii) 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}. (iii) 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 proof of Theorem 3.15.

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 $(P_1|| \ldots ||P_N) = (V, L, InitL, A, PV, PA)$, where V, L, $InitL$, PV, and PA are as defined in Definition 3.11, and:

- A consists of the following types of actions:
  - Actions corresponding to actions in A_j, for some j, as defined in Definition 3.11.
  - Actions that correspond to the initiation and termination of the components of the composition: (i) 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. (ii) 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.

□

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 that $(P_1|| \ldots ||P_N) \sim (P_1; \ldots; P_N)$. 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: (i) Execution of b does not affect (in the sense of Definition 3.7) whether a is enabled, and vice versa. (ii) 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. (Observe that if a and b are nondeterministic, there may be more than one such state s_2.) □

*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 $\sim$ sequential for **arb***-compatible programs).* If $P_1, \ldots, P_N$ are **arb**-compatible, then $(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N)$. □

A detailed proof of this theorem is given in [15]. The proof relies on showing that for every maximal computation of the parallel composition there is a computation of the sequential composition that is equivalent with respect to the nonlocal variables, and vice versa.

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. Examples of valid and invalid uses of **arb** composition are given in [15].

We also define an additional bit of syntactic sugar useful in improving the readability of nestings of sequential and **arb** composition: $\mathbf{seq}(P_1, \ldots, P_N) \sim (P_1; \ldots; P_N)$.

arb composition has a number of useful properties. It is associative and commutative (proofs given in [15]), and it allows refinement by parts, as the following theorem states. This last property is the justification for our program-development strategy, in which we apply the techniques of sequential stepwise refinement to **arb**-model programs.

Theorem 3.16 (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 $\mathbf{arb}(P_1, \ldots, P_N) \sqsubseteq \mathbf{arb}(P'_1, \ldots, P'_N)$. □

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 3.17 (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 3.18 (***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. □

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, unlike our operational model's semantics, permit aliasing. Syntactic restrictions sufficient to guarantee **arb**-compatibility do not seem in general feasible. However, we can 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 3.18. In [15] we discuss such rules for two representative programming notations and present examples of their use.

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. **arb**-model programs are easily converted into sequential programs in the underlying sequential notation by replacing **arb** composition with sequential composition; simple syntactic transformations also suffice to convert them to a notation that includes a construct implementing general parallel composition as defined in Definition 3.12. Details and examples are given in [15].

4 The par model and shared-memory programs

As discussed in Section 2, once we have developed a program in our **arb** model, we can transform it 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 this section we sketch how we extend our model of parallel composition to include barrier synchronization, give key transformations for turning **arb**-model programs into **par**-model programs, and briefly discuss executing such programs on shared-memory architectures. Details are given in [15].

A preliminary remark: 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.

Parallel composition with barrier synchronization. We first expand the definition of parallel composition given previously to include barrier synchronization, a common form of synchronization (see [1], e.g.) based on the notion of a "barrier" that all processes much reach before any can proceed. (A formal specification is given in [15].)

We define barrier synchronization in terms of our model by defining a new command, **barrier**, and extending Definition 3.12. This combined definition (presented in detail in [15]) implements a common approach to barrier synchronization based on keeping a count of processes waiting at the barrier. 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. This behavior is implemented by the protocol actions of the **barrier** command.

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, 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 **par**-compatible elements "match up" with regard to their use of the **barrier** command — that is, they all execute **barrier** the same number of times and hence do not deadlock. (For example, $(a := 1; \mathbf{barrier}; b' := a)$ and $(b := 2; \mathbf{barrier}; a' := b)$ are **par**-compatible, but they would not be if the **barrier** command were removed from one of them.) A detailed definition of **par**-compatibility, based on our model, is given in [15]. An aspect that should be noted here is that part of the definition involves extending the definition of **arb**-compatibility to require that **arb**-compatible elements not interact with each other via barrier synchronization.

As with **arb**, we write $\mathbf{par}(P_1, \ldots, P_N)$ to denote the parallel composition (with barrier synchronization) of **par**-compatible elements $P_1, \ldots, P_N$. Examples of valid and invalid uses of **par** composition are given in [15].

Transforming arb-model programs into par-model programs. The transformation of **arb**-model programs into **par**-model programs involves a relatively small collection of transformations that describe how to "interchange" **arb** composition with the standard constructors of the sequential model, namely sequential composition and the *IF* and *DO* constructs of Dijkstra's guarded-command language [8]. (The ideas are not specific to Dijkstra's guarded-command language; analogous transformations can be defined for any notation that provides equivalent constructs.) Space limitations preclude presenting all the transformations in this paper; we present one of the simpler transformations in this section and note that the full set of transformations is given in [15] along with examples of their use.

Theorem 4.1 (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

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

□

Executing par-model programs. **par** composition as described in this section is implemented by general parallel composition (as described in Section 3) plus a barrier synchronization that meets the specification of [15]. Thus, we can transform a program in the **par** model into an equivalent program in any language that includes 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). Details and examples are given in [15].

5 The subset par model and distributed-memory programs

As discussed in Section 2, once we have developed a program in our **arb** model, we can transform it 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 this section we sketch how 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.

Parallel composition with message-passing. We first expand the definition of parallel composition given previously to include message-passing over single-sender–single-receiver channels with infinite slack (defined much as in [1]; a formal specification is given in [15].)

We define message-passing in terms of our model by defining two new commands, **send** and **recv**, and extending Definition 3.12. This combined definition (presented in detail in [15]), like many other implementations of message-passing, represents channels as queues, with message sends and receives modeled as enqueue and dequeue operations. The **send** command includes local protocol variables of type Queue ("outports") and a protocol action that performs an enqueue on one of them; the **recv** command includes local protocol variables of type Queue ("inports") and a protocol action that performs a dequeue on one of them (suspending until the queue is nonempty). (As in

Section 4, we model suspension as busy waiting.) The extended definition of parallel composition includes local protocol variables of type Queue ("channels"), one for each ordered pair of component programs (elements of the composition), and specifies that as part of the composition operation the component programs P_j are modified by replacing inport and output variables with channel variables. That is, the composition "connects" the inports and outports to form channels.

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. We refer to the former as *local-computation sections* and to the latter (together with the preceding and succeeding barrier synchronizations) as *data-exchange operations*.

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: (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 $W_1, \ldots, W_N$ (sets of "local data" as mentioned previously); and (iii) the forms of $P_1, \ldots, P_N$ are such that they "match up" with respect to execution of barriers (as for **par**-compatibility), with the additional restriction that each group of "matching" sections (between barriers) constitutes either a local-computation section or a data-exchange operation. Details and examples are given in [15].

Transforming subset-par-model programs into message-passing programs. 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: (i) Map each element W_j of the partition of V to the address space for process j. (ii) Convert each data-exchange operation (consisting of a set of $(\mathbf{barrier}; Q'_j; \mathbf{barrier})$ 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. (iii) 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.

Such a program refines the original program, as discussed in [15]. An example of applying these transformations is also given in [15].

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. Details and examples are given in [15].

6 Related work

Other researchers, for example Back [2] and Martin [14], 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).

Our operational model is based on defining programs as state-transition systems, as in the work of Chandy and Misra [5], Lamport [12], Manna and Pnueli [13], and others. Our model is designed to be as simple as possible while retaining enough generality to support all aspects of our programming model.

Programming models similar in spirit to ours have been proposed by Valiant [17] and Thornley [16]; our model differs in that we provide a more explicit supporting theoretical framework and in the use we make of archetypes.

Our work is in many respects complementary to efforts to develop parallelizing compilers, for example Fortran D [7]. 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.

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 [6] and Brinch Hansen [4]. 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, presented in detail in [15], 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 [15] 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 [15] 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 [18] that was an early inspiration for this work.

References

1. G. R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc., 1991.
2. 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.
3. 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.
4. P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.
5. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
6. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
7. 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.
8. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
12. L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
13. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.
14. A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
15. B. L. Massingill. A structured approach to parallel programming. Technical Report CS-TR-98-04, California Institute of Technology, 1998. Ph.D. thesis.
16. J. Thornley. A parallel programming model with sequential semantics. Technical Report CS-TR-96-12, California Institute of Technology, 1996.
17. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
18. E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.