Parallel Programming: Achieving Portability Through Abstraction

Ying Liu* and Ambuj K. Singh*
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

Abstract
Translating parallel programs across different architectures is a major problem in parallel programming. Part of this difficulty stems from the fact that the synchrony structure of a program is fixed. It is not at all obvious how the synchrony structure of a program can be modified in order to maximize efficiency while preserving correctness. We propose the idea of abstract programs to solve this problem. Abstract programs have a flexible synchrony structure and therefore, can be easily mapped across architectures.

1 Introduction
Parallel programming poses a challenge for the design of programming languages and programming methodologies. Historically, there have been different design methodologies for each architecture (SIMD vs. MIMD, shared-variable vs. message-passing) in order to maximize the efficiency of executing programs. But, this efficiency has been achieved at the expense of portability across architectures. This realization has prompted the search for design tools and abstractions which are independent of any one architecture, but efficiently implementable across many architectures [8]. The Unity model of parallel programming [1] that advocates the unity of the programming task is such an abstraction. The salient features of Unity are the presence of nondeterminism and the absence of control structures. A Unity program consists of a set of multiple-assignment statements and a program execution consists of any fair selection of the statements. Statements grouped in the same multiple assignment statement are executed synchronously, whereas statements grouped in different multiple assignment statements are executed asynchronously. The simple structure of Unity programs along with an elegant characterization of the synchrony and asynchrony of programs is instrumental in the development of concise and clear programs for a wide variety of architectures [1].

Stepwise development [2, 4, 7] encourages programming in which much of a program's derivation is independent of any particular architecture; thus, alternatives can be retargeted during the later refinement stages without substantial program modification [3]. Stepwise development in Unity consists of two phases: property refinement and program refinement [1]. During property refinement, a property of the specification is successively refined to a stronger property that is easily implementable. Property refinement continues until it is clear as to how a program meeting the refined specification can be drawn up. This program forms the starting point for program refinement. During program refinement, a program is continuously refined until we obtain a complete running program on the target architecture viz. asynchronous shared-memory architecture. The issues addressed by program refinement include implementation of abstract data types, division of program into processes, and interprocess synchronization [6]. Both of the above mentioned refinements are useful in the development of programs; property refinement is usually domain-specific, whereas program refinement is much more general-purpose in nature.

Although Unity theory has been very successful in supporting property refinement, programs derived by property refinement (which form the starting point for program refinement) are sometimes difficult to map onto target architectures. This is because of a key difference between property refinement and program refinement: property refinement preserves only the specified properties while program refinement preserves all properties of the program (with no regard to the initial specification for which the program was written). Thus, when we fix on a particular Unity program, we also fix on a particular synchrony relation among the the basic assignment actions; actions gathered into the same multiple-assignment statement can only be
executed synchronously whereas actions in different multiple-assignment statements can only be executed asynchronously. The stratification of the synchrony relation has an adverse effect on the later program refinements as they now have to preserve a number of unwanted properties. Consequently, the refinements tend to be somewhat limited and implementations across different architectures share little of the program refinement phase of stepwise development.

To avoid the drawbacks of stratifying a particular synchrony structure, we introduce the concept of an abstract program. An abstract program has a set of basic statements and an asynchrony relation over the basic statements. The asynchrony relation describes the restriction on the basic statement executions: if two basic statements are constrained by the asynchrony relation, they can only executed asynchronously; otherwise, they can be executed either synchronously or asynchronously. This under-specification of the synchrony structure allows program refinements for different architectures to share the initial steps and provides a basis for a much more flexible and portable program design.

The idea of abstract programs is partly motivated by the idea of synchronous groups proposed by Roman and Cunningham [5]. In their formalism, a program contains a set of transactions and a synchronous group is defined to be a subset of transactions that are executed synchronously. A synchronous group is different from a statement in Unity, as it can be modified dynamically. Thus, synchronous groups facilitate a programming style in which the granularity of the computation can be changed to accommodate structural variations. In spite of its advantages, the idea of synchronous groups does not allow for under-specification. Like a Unity program, a program in [5] has a completely specified synchrony structure; it is not possible to delay the specifics of statement composition until the target architecture is completely known.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the Unity syntax and logic. In Section 3, we present the idea of abstract programs. In Section 4, we use two simple examples: sorting an array and summing an array, to illustrate the usefulness of abstract programs. Finally, Section 5 includes concluding remarks.

2 A Brief Introduction to Unity

A Unity program consists of four sections: a declare section that declares the variables used in the program, an always section that consists of a set of proper equations defining certain variables as functions of other variables, an initially section that describes the initial values of the variables, and an assign section that consists of a non-empty set of assignment statements.

An assignment statement contains one or more assignment components separated by \|
. An assignment component is either a quantified assignment or an enumerated assignment. A quantified assignment specifies a quantification and an assignment that is to be instantiated with the given quantification; a quantification names a set of bound variables and a boolean expression satisfied by the instances of the bound variables. An enumerated assignment has a variable list on the left, a corresponding expression list in the middle, and a boolean expression on the right:

\[(\text{variable-list}) := (\text{expression-list}) \text{ if } (\text{condition}).\]

An assignment component is executed by first evaluating all expressions and then assigning the values of evaluated expressions to the appropriate variables, if the associated boolean is true; otherwise, the variables are left unchanged.

The set of assignment statements in assign section is written down either by enumerating every statement singly and using \| as the set constructor, or by using a quantification of the form \( (\forall \var : \text{range} : \text{statement}).\)

A program execution starts from any state satisfying the initial conditions and goes on forever; in each step of execution some assignment statement is selected non-deterministically and executed. Non-deterministic selection is constrained by the following fairness rule: every statement is selected infinitely often [1].

There are five relations on predicates in Unity theory: unless, stable, invariant, ensures, and leads-to. The first three are used for stating safety properties whereas the last two are used for stating progress properties. In the following description, \( s \) is an arbitrary statement in a given program and \( p, q \) are predicates.

- \( p \text{ unless } q \equiv (\forall s :: (p \land \neg q) \land (p \land q)).\)
- \( \text{stable } p \equiv p \text{ unless false}.\)
- \( \text{invariant } p \equiv \text{initially } p \land p \text{ unless false}.\)
- \( \text{p ensures } q \equiv p \text{ unless } q \land (\exists s :: (p \land \neg q) \land (q)).\)
- \( \text{The relation leads-to, denoted by } \Rightarrow, \text{ is defined by following three rules.}\)
  1. \( p \text{ ensures } q \Rightarrow p \Rightarrow q.\)
  2. \( (p \Rightarrow q \land q \Rightarrow r) \Rightarrow p \Rightarrow r, \text{ and}\)
  3. \( \text{For any set } W, (\forall m : m \in W : p.m \Rightarrow q) \Rightarrow ((\exists m : m \in W : p.m) \Rightarrow q).\)
The fixed point of a program, represented by \( FP \), describes the state of the program upon termination. It is defined to be the conjunction of the fixed point of the statements of the program. The fixed point of a statement is in turn obtained by replacing the assignment symbol \( := \) by the equality symbol \( = \). In other words, for any program \( P \),

- \( FP-P \equiv (\forall s : s \in P : FP-s) \),

where for any \( s :: X \Leftarrow E \), \( FP-s \equiv B \Rightarrow (X \Leftarrow E) \).

Next, we discuss the idea of program refinements in Unity. There are two kinds of program refinements that are of interest: those that preserve all unless and leads-to properties, and those that preserve the fixed point. The first kind of refinement is called a property preserving refinement and the second kind is called a fixed point preserving refinement. We say that a program \( G \) is a property preserving refinement of a program \( F \) if and only if for all predicates \( p, q \) the following two assertions hold.

- \( p \text{ unless } q \text{ in } F \Rightarrow p \text{ unless } q \text{ in } G, \)
- \( p \text{ leads to } q \text{ in } F \Rightarrow p \text{ leads to } q \text{ in } G. \)

We say that \( G \) is a fixed point preserving refinement iff the following two assertions hold.

- \( FP-G \Rightarrow FP-F, \)
- \( \text{true} \Rightarrow FP-F \text{ in } F \Rightarrow \text{true} \Rightarrow FP-G \text{ in } G. \)

3 Abstract Program

An abstract program \( F \) consists of a basic statement set \( S \) that describes the basic actions and an irreflexive, symmetric asynchrony relation \( A \) that constrains the execution of the statements in \( S \). Each basic statement is an assignment statement of the form defined in the previous section. Relation \( A \) over \( S \) specifies the asynchrony relationship of the basic statements: \( A(s_1, s_2) \) implies that \( s_1, s_2 \) can only be executed asynchronously. If \( A(s_1, s_2) \) does not hold, the \( s_1, s_2 \) can be executed in any order, either \( s_1 \parallel s_2 \) or \( s_1 \parallel s_2 \). An abstract program has no fixed synchrony relation; it advocates flexibility of program structure.

Given an abstract program \( F = (S, A) \), a compound statement \( v \) of \( F \) is defined to be a multiple assignment statement composed of the basic statements and complying with the restrictions of \( A \). In other words, \( v \) is a compound statement of \( F \) iff for any pair of basic statements \( s_1, s_2 \) in \( v \), there is no asynchrony relation between them, i.e., \( \neg A(s_1, s_2) \). We use \( [v]F \) to denote the set of basic assignment statements constituting \( v \), i.e.,

\[
[v]F \equiv \{ s : s \in S \land s \text{ is a component of } v \}.
\]

The fixed point of a compound statement is defined to be the conjunction of the fixed points of the component statements.

As an example, consider an abstract program \( F \) with \( S = \{ s_1, s_2, s_3, s_4 \} \) and \( A = \{ (s_1, s_2) \} \). Then \( v = s_1 \parallel s_2 \parallel s_3 \) is a compound statement of \( F \), and \( [v]F = \{ s_1, s_2, s_3 \} \). The fixed point of compound statement \( v \) is the conjunction of the fixed points of \( s_1, s_2, \) and \( s_3 \). Compound statement \( v' = s_1 \parallel s_2 \parallel s_4 \) is an invalid compound statement as \( A(s_1, s_4) \) holds in \( F \). Note that a Unity program is an abstract program where each statement is a basic statement and all statements are restricted to be executed asynchronously (i.e., the asynchrony relation contains all possible pairs).

We define \( \mathcal{M}(F) \) to be the set of all possible compound statements abstract program \( F \), i.e.,

\[
\mathcal{M}(F) \equiv \{ v : v \text{ is a compound statement of } F \}.
\]

Observe that for a Unity program \( F = (S, A) \),

\[
\mathcal{M}(F) = S.
\]

As mentioned earlier, a number of design decisions concerning the target architecture have to be made during program refinement. One of these decisions involves refining the synchrony relation of the program to suit the target architecture. The idea of implementation of abstract programs defined below provides us with the necessary theory for this kind of refinement. Given an abstract program \( F = (S, A) \), we say that another abstract program \( F' = (S', A') \) is an implementation of \( F \) if and only if the following three conditions hold.

1. The set of basic statements in \( S' \) are obtained from the compound statements of \( F \), i.e., \( S' \subseteq \mathcal{M}(F) \).
2. Every basic statement in \( S \) is included in some basic statement in \( F' \), i.e.,
   \[
   (\forall s : s \in S : (\exists t : t \in S' : s \in [t]F')).
   \]
3. The asynchrony relation of \( F' \) obeys the asynchrony relation of \( F \), i.e.,
   \[
   (\forall s_1, s_2, t_1, t_2 : s_1, s_2 \in S' \land t_1, t_2 \in S' \land
   s_1 \in [t_1]F' \land s_2 \in [t_2]F' : (s_1, s_2) \in A \Rightarrow (t_1, t_2) \in A').
   \]

In other words, abstract program \( F' \) contains the same set of assignments as abstract program \( F \), but is possibly more specific about the asynchrony relation among the statements. Observe that if \( F' \) is an implementation of \( F \), then \( \mathcal{M}(F') \subseteq \mathcal{M}(F) \). Moreover, \( S = \bigcup_{E \in S'} [t]F' \).
Now, we extend the logic of Unity to accommodate abstract programs. We redefine the primary relations unless, ensures, and the fixed point for an abstract program $F = (S, A)$. The secondary relations, such as stable, invariant, and leads-to, are defined as in Section 2.

- $p$ unless $q$ $\equiv$ $(\forall t : t \in \mathcal{M}(F) : \{p \land \neg q\} \vee \{p \lor q\})$
- $p$ ensures $q$ $\equiv$ $(\exists s : s \in S : (\forall v : v \in \mathcal{M}(F) \land s \in [v]_F : \{p \land \neg q\} \lor \{q\}))$
- $FP-F$ $\equiv$ $(\forall s : s \in S : \text{FP-s})$

Based on the above definitions, we prove the following theorem.

**Theorem:** Any implementation $F' = (S', A')$ of an abstract program $F = (S, A)$ is a proper and fixed point preserving refinement.

**Proof:**

- $p$ unless $q$ in $F$ $\Rightarrow$ $p$ unless $q$ in $F'$
  
  $p$ unless $q$ in $F$
  $\equiv$ $(\forall t : t \in \mathcal{M}(F) : \{p \land \neg q\} \lor \{p \lor q\})$
  $\Rightarrow$ $(\forall t : t \in \mathcal{M}(F') : \{p \land \neg q\} \lor \{p \lor q\})$
  $\equiv$ $(\forall t : t \in \mathcal{M}(F') : \{p \land \neg q\} \lor \{p \lor q\})$

  - $p$ ensures $q$ in $F$ $\Rightarrow$ $p$ ensures $q$ in $F'$
    
    $p$ ensures $q$ in $F$
    $\equiv$ $(\exists s : s \in S : (\forall v : v \in \mathcal{M}(F) \land s \in [v]_F : \{p \land \neg q\} \lor \{q\}))$
    $\Rightarrow$ $(\exists s : s \in S' : (\forall v : v \in \mathcal{M}(F') \land s \in [v]_{F'} : \{p \land \neg q\} \lor \{q\}))$

  - $FP-F'$ $\equiv$ $FP-F$

**FP-F'**

- $FP-F'$$\equiv (\forall s : s \in S' : \text{FP-s})$

Observe that all theorems about unless (and consequently, stable and invariant), ensures, lead-to, and the fixed point in [1] also hold for abstract programs.

4 Examples

In this section, we illustrate the usefulness of abstract programs through two examples. The first example is based on sorting an array, and the second is based on summing an array.

4.1 Sorting an array

In this problem we are given an input array $X[1..N]$ that is to be sorted in an increasing order by exchange sort and the result has to be stored in an output array $y[1..N]$. In our algorithm, array $y$ is initialized to array $X$, and any two neighboring elements in $y$ that are not in an increasing order are exchanged until the array is sorted. The following is the formal specification of the solution strategy as developed in [1].

1. **Invariant** $y$ is a permutation of $X$
2. $FP$ $\equiv$ $(M = 0)$
3. $(\forall k : k > 0 : M = k \rightarrow M < k)$
The introduction of an abstract program overcomes these problems since its flexible synchrony structure avoids over-specification. We illustrate this by designing an abstract program \( F = (S, A) \) for the above sorting problem. From the specification, a basic statement set \( S \) and an asynchrony relation \( A \) may be written down as follows.

\[
S = \{ t_i : 1 \leq i < N \}, \quad A = \{(t_i, t_j) : |i - j| = 1 \}
\]

Where \( t_i : y[i], y[i + 1] := \text{sort}'(y[i], y[i + 1]) \).

As shown below, the proof of correctness of \( F \) is not any more difficult than the proof of \( P \) or \( Q \). The proof obligation is to show that \( F \) satisfies the conditions (1), (2), and (3).

\[
\text{Proof of (1): Straightforward and omitted.}
\]

\[
\text{Proof of (2):}
\]

\[
FP \cdot F
\]

\[
\equiv \{ \text{definition of fixed point} \}
\]

\[
(V i : 1 \leq i < N : y[i], y[i + 1] = \text{sort}'(y[i], y[i + 1]) \}
\]

\[
\equiv \{ \text{definition of sort}' \}
\]

\[
(V i : 1 \leq i < N : y[i] \leq y[i + 1]) \equiv \{ \text{definition of } M \}
\]

\[
M = 0
\]

\[
\text{Proof of (3): It is straightforward to see that the metric } M \text{ never increases and that the reordering of any two neighboring elements can only decrease it. Therefore,}
\]

\[
M = k \land y[i] > y[i + 1] \text{ unless } M < k.
\]

Let \( i \) be any index for which the neighbors \( y[i] \) and \( y[i + 1] \) are not in the right order, i.e., \( y[i] > y[i + 1] \). Consider the basic statement \( t_i \) that exchanges these two elements and any compound statement \( r \) containing \( t_i \) (i.e., \( r = t_i \mid \ldots \)). Then, the execution of \( r \) will interchange \( y[i] \) and \( y[i + 1] \) and reduce the metric, i.e.,

\[
(M = k \land y[i] > y[i + 1]) \quad \text{or } \quad \{M < k\}
\]

\[
(\exists i: 1 \leq i < N : y[i] > y[i + 1]) \quad \text{unless } M < k
\]

\[
M = k \land y[i] > y[i + 1] \text{ ensures } M < k
\]

\[
(\exists i: 1 \leq i < N : y[i] > y[i + 1]) \quad \text{and taking disjunction}
\]

\[
M = k \land k > 0 \Rightarrow (\exists i: 1 \leq i < N : y[i] > y[i + 1]) \quad \text{and taking disjunction}
\]

\[
M = k \land k > 0 \Rightarrow M < k.
\]

This concludes the proof of correctness of the proposed abstract program \( F \). Program \( F \) can be easily implemented on a variety of architectures and represents the
abstraction of a number of Unity programs. In particular, programs $P$ and $Q$ are easily seen to be implementations of $F$.

4.2 Summing an array

In this problem we are given an input array $X[l..N]$ that is to be summed. In order to simplify the presentation, we assume $N$ is a power of 2, i.e., $N = 2^n$ for some positive integer $n$. The algorithm taken from [5] uses a temporary array $y[l..N]$ that is initialized to the input array $X$, and repeatedly sums selected elements in $y$ that are $2^i$ distance away for $i = 0, \ldots, n-1$. In the $i$th step, elements $y[k]$ and $y[k + 2^i]$, where $k \mod 2^{i+1} = 1$, are added and the result is stored in $y[k]$. The algorithm terminates after $n$ steps when the sum of the entire array is stored in $y[1]$. Figure 1 shows the summation strategy for $n = 4$.

We skip the specification and the correctness proof and focus on the flexibility of implementation achieved by using an abstract program. In designing the basic statement set $S$ for this problem, observe that there are $N-1$ additions, each of which is executed once. Therefore, it is natural to adopt each addition as a basic statement. Let $s_{i,j}$ denote the basic statement corresponding to the addition of $y[i]$ and $y[i + j]$. Observe that the addition in $s_{i,j}$ cannot be performed until the additions in $s_{i,j/2}$ and $s_{i,j/2 + j}$ are completed. (For example, in Figure 1, the execution of $s_{1,2}$ must follow the executions of $s_{1,1}$ and $s_{3,1}$.) To meet this requirement, we introduce a group of control variables $b(i,j), j > 0$ for every basic statement $s_{i,j}$ and $N$ additional variables $b(i,0)$. Initially, each $b(i,j), j > 0$ is set to false and each $b(i,0)$ is set to true. Statement set $S$ is defined as follows:

$$S = \{s_{i,j} : (3k < k \leq n - 1 : j = 2^k \land i \mod 2^{k+1} = 1 \land i \leq 2^n)\}$$

where statement $s_{i,j}$ is defined to be

$$y[i], b(i,j), b(i,j/2), b(i + j,j/2) := y[i] + y[i + j], true, false, false \text{ if } b(i,j/2) \land b(i + j,j/2).$$

Asynchrony relation $A$ constrains the basic statements that access common variables to be executed asynchronously, i.e.,

$$A = \{(s_{i,j}, s_{k,l}) : (k \leq i \land i + 2j \leq k + 2l) \lor (i \leq k \land k + 2l \leq i + 2j)\}$$

This completes the description of the abstract program $F = (S,A)$. Next, we consider an implementation of $F$ when the target machine has three synchronous processors. For simplicity, assume that $n = 4$, i.e., the abstract program corresponds to Figure 1. We define an implementation $G$ as follows. Because there are three processors, we define every statement in $G$ to be consisting of at most three statements from $F$, one for each processor. As every addition is executed only once, every statement in $F$ appears exactly once in the statements of $G$. The following program $G$ is easily seen to be an implementation of program $F$.

Program $G$

assign

$$s_{1,1} \rightarrow s_{3,1} \rightarrow s_{5,1}$$

$$s_{7,1} \rightarrow s_{9,1} \rightarrow s_{11,1}$$

$$s_{13,1} \rightarrow s_{15,1} \rightarrow s_{12}$$

$$s_{5,2} \rightarrow s_{9,2} \rightarrow s_{13,2}$$

$$s_{14} \rightarrow s_{9,4}$$

$$s_{8,8}$$

end

Observe that we are able to take advantage of the number of processors to refine the synchrony structure. If the number of processors now changes to four, we can immediately write the following program $G'$ by refining abstract program $F$.

Program $G'$

assign

$$s_{1,1} \rightarrow s_{3,1} \rightarrow s_{5,1} \rightarrow s_{7,1}$$

$$s_{9,1} \rightarrow s_{11,1} \rightarrow s_{13,1} \rightarrow s_{15,1}$$

$$s_{12} \rightarrow s_{5,2} \rightarrow s_{9,2} \rightarrow s_{13,2}$$

$$s_{14} \rightarrow s_{9,4}$$

$$s_{8,8}$$

end

This illustrates how the flexibility of abstract programs can be put to good use in the formal development of portable programs.

5 Discussion

The proper structure of a program plays an important role in refinements of the program to a target machine. We found that the fixed-structure of Unity programs leads to an overspecification that has an adverse affect on the applicability of program refinements. In order to solve this problem, we propose the idea of abstract programs. Abstract programs have a flexible synchrony structure and allow decisions about the actual structure of a program to be delayed until the target architecture...
is completely known. This permits programs for different architectures to share the initial steps of program development and thus makes programs more portable across architectures. Fortunately, abstract programs share much of the Unity logic and proof system. Thus, all the ideas, methodologies, and proofs presented in [1] are also applicable to abstract programs. It should be pointed out that other refinements addressing synchronization, implementation of abstract date types, and scheduling of processes [6] are still necessary in the phase of program refinement. However, we do not think the introduction of abstract programs makes these any more difficult.

References


---

Figure 1. Summing an Array of 16 Elements