# Developing Parallel Programs: A Design-Oriented Perspective

Ali Ebnenasir
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

Rasoul Beik
Computer Engineering Department
Islamic Azad University
Khomeini Shahr, Isfahan, Iran
beik@iaukhsh.ac.ir

## Abstract

*The behavioral correctness of parallel programs has a pivotal role in computational sciences and engineering applications as researchers draw scientific conclusions from the results generated by parallel applications. Moreover, with the advent of multicore processors, the development of parallel programs should be facilitated for the mainstream developers. While numerous programming models and APIs exist for parallel programming, we pose the view that more emphasis should be placed on designing the synchronization mechanisms of parallel programs independent from the design of their functional behaviors. More importantly, programs' behaviors evolve (due to new requirements and change of configuration), thereby creating a need for techniques and tools that enable developers to reason about the behavioral evolution of parallel programs. With such motivations, we introduce a framework for automated design/evolution of the synchronization mechanisms of parallel programs.*

## 1 Introduction

Developing applications for parallel platforms is one of the major challenges for mainstream developers and the software engineering community in the coming decade. While numerous programming models, Application Programming Interfaces (APIs) and parallel design patterns exist for parallel programming and parallelization of sequential programs, the complexity of designing *correct* parallel programs has continuously been underestimated; existing programming models/APIs lack the necessary machinery that enables developers to reason about the behaviors of the systems they produce. In this paper, we present a platform-independent framework for automated design of the *synchronization skeleton* of parallel programs, where the synchronization skeleton of a program includes only the parts of its functionalities related to inter-thread[1] synchronization. More importantly, upon the detection of new requirements, our framework automatically determines whether or not an existing synchronization skeleton can be revised so it captures the new requirements.

Numerous approaches exist for parallel programming most of which provide programming models and linguistic structures [25, 13, 36, 22, 37, 24, 7, 15], function libraries and run-time systems [11, 44] and programming patterns [38, 1] to enable developers in *creating* parallel programs while lacking the necessary infrastructure to facilitate *reasoning* about behavioral correctness of parallel programs. For example, Pthreads [24] provides necessary data structures and function libraries (in C/C++) to enable programmers for explicit multithreading, where programmers have full control over creating, synchronizing and destroying concurrent threads of execution. OpenMP [37, 8] is an API that augments C/C++ and Fortran with a set of compiler directives, run-time library routines and environment variables to enable parallel programming in a shared memory model on multiple platforms. Intel Threading Building Blocks [44] provides a task-based model for parallel programming that abstracts away many thread management issues with the help of a task manager that analyzes the underlying platform to balance the work load over available processors. Microsoft Task Parallel Library (TPL) [35] provides a class library for expressing potential parallelism and enabling dynamic task distribution. While the aforementioned approaches facilitate the creation of parallel applications, they often lack the necessary techniques/tools for reasoning about the behaviors of parallel programs in the design phase.

We present our position that designing parallel programs in an abstract computational model deserves more attention from the research community. Such a design-oriented development method enables automated reasoning about

[1]In this paper, we use the terms *thread* and *process* interchangeably.

inter-thread synchronization. To further support our position, we quote a statement from Asanovic *et al.* [6] (Page 7 Section 3) as follows:

"$\cdots$ *it seems unwise to let a set of existing source code drive an investigation into parallel computing. There is a need to find a higher level of abstraction for reasoning about parallel application requirements.*"

In this paper, we first present an overview of our proposed design method in Section 2. We then demonstrate our proposed approach in the context of a barrier synchronization protocol in Section 3. Subsequently, in Section 4, we discuss related work. Finally, we make concluding remarks in Section 5.

## 2 Proposed Approach: Property-Oriented Design

Our focus is on the following *redesign problem*:

*when a program fails to satisfy a new global requirement (i.e., property) such as mutual exclusion, deadlock-freedom and progress (i.e., guaranteed service), how should the local synchronization mechanism of each thread be revised so the global requirement is met*?

This is an important problem in developing and maintaining parallel applications. An example of such requirements is a case where private data structures have to be shared (possibly for performance improvement purposes). In principle, such cases are instances of adding new safety (respectively, mutual exclusion) requirements to the set of program requirements. Another example is a case where an existing program should meet a new progress requirement and designers must determine what local changes should be made in the synchronization skeleton of each thread such that the entire program meets the new progress requirement while preserving its existing requirements. While in our previous work, we have illustrated that, in general, redesigning parallel programs for multiple progress properties is a hard problem [19], the focus of our current research is on (1) identifying cases where the redesign problem can be solved efficiently, (2) providing automated support for redesign in a high-level model of parallel computation. In this section, we propose a property-oriented technique focused on stepwise (re)design of the synchronization skeleton of parallel programs, where a *synchronization skeleton* is an abstraction of the parts of the program functionalities that handle inter-thread synchronization.

In our approach, we separate the high-level design of inter-thread synchronization mechanisms from the design and implementation of functional requirements. Figure 1 illustrates that, in our proposed approach, we start with an input synchronization skeleton (i.e., design) and a desired property that should be captured by an evolved version of the input design. For example, the input design may be the synchronization skeleton of a mutual exclusion program that guarantees mutually exclusive access to a critical section, but does not guarantee progress for some threads, where each thread should eventually get a chance to access the critical section. The input design can be created manually. In other words, the input design is the first educated guess of the designer for a program that meets all its requirements. We specify/design synchronization skeletons in a *shared memory model* and a *high atomicity program model* in which threads can read/write several program variables in an atomic step. The motivation behind this abstract memory/program model is to (1) simplify the design for mainstream developers; (2) develop efficient automated design algorithms since the complexity of automation increases as we consider additional constraints [16], and (3) potentially reduce development costs by identifying the impossibility of revising an input design towards satisfying a set of desired global properties as early as possible; if a program cannot be revised in a high atomicity program model, then it cannot be modified under additional constraints of a more concrete program model as well (See [31] for proof). The third motivation is actually due to the fact that sometimes designers spend time on fixing an existing design without knowing that the design at hand is not fixable. We have developed sound and complete redesign algorithms [19, 17] that enable developers to detect such impossibility of redesign. Next, we present an overview of our redesign algorithms and the way program properties should be specified.

**Overview of the property-oriented (re)design algorithm.** Consider a case where a thread $\mathcal{T}$ of a program $p$ does not meet a required progress property $\mathcal{L}$ that states *it is always the case that if $\mathcal{T}$ is trying to access a shared resource, then $\mathcal{T}$ will eventually get access to the shared resource*. The input to our redesign algorithm is a synchronization skeleton of $p$ in terms of Dijkstra's guarded commands (actions) [14] and the property $\mathcal{L}$ specified in terms of Linear Temporal Logic (LTL) [21] properties. A guarded command is of the form $grd \rightarrow stmt$, where the guard $grd$ is a Boolean expression specified in terms of program variables and the statement $stmt$ denotes a set of assignments that *atomically* update program variables when the $grd$ holds. The LTL properties can be extracted from English statements of such properties using existing techniques [29]. For example, a LTL expression representing the above progress property is as follows: $\Box$ (( $\mathcal{T}$ is trying to access a shared resource) $\Rightarrow$ $\Diamond$ ($\mathcal{T}$ has access to the shared resource)), where $\Box$ denotes the temporal operator *always*, $\Diamond$ represents the *eventuality* and $\Rightarrow$ is the propositional implication. Our objective is to synthesize a revised version of $p$, denoted $p_r$, that meets

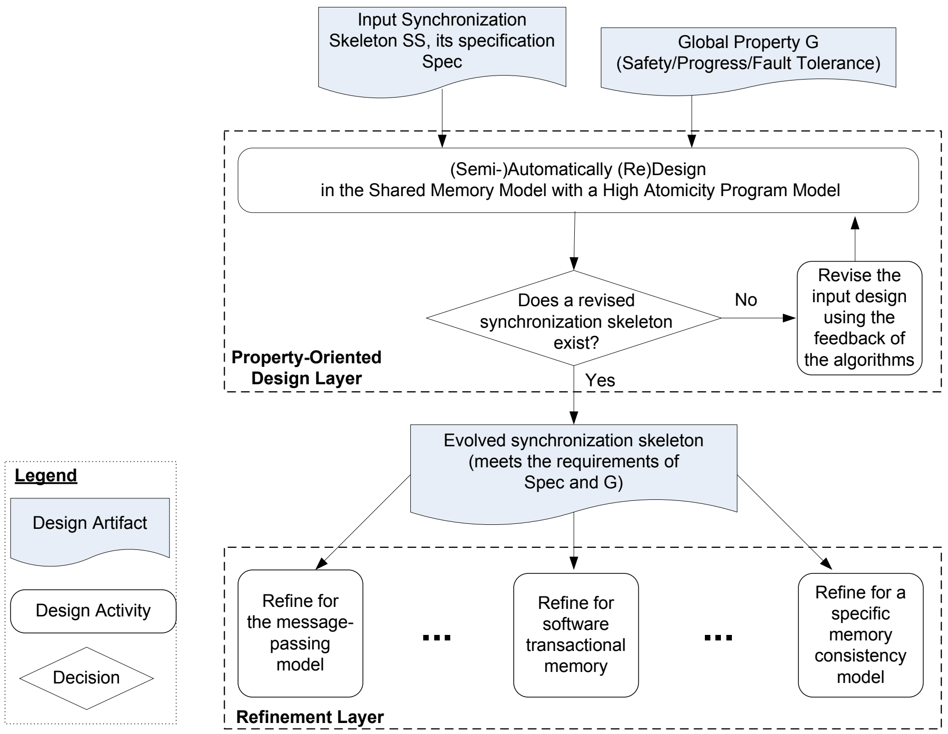
Input Synchronization Skeleton SS, its specification Spec
Global Property G (Safety/Progress/Fault Tolerance)
(Semi-)Automatically (Re)Design in the Shared Memory Model with a High Atomicity Program Model
Does a revised synchronization skeleton exist?
No
Revise the input design using the feedback of the algorithms
Yes
Property-Oriented Design Layer
Evolved synchronization skeleton (meets the requirements of Spec and G)
Legend
Design Artifact
Design Activity
Decision
Refine for the message-passing model
...
Refine for software transactional memory
...
Refine for a specific memory consistency model
Refinement Layer

**Figure 1.** A framework for property-oriented design and evolution of the synchronization skeleton of parallel programs.

$\mathcal{L}$ for $\mathcal{T}$ and preserves all correctness properties of $p$. To achieve this goal, the redesign algorithm performs the following steps on a *reachability graph* of the input program. The reachability graph is a state transition graph created by exploring all possible states reachable from program initial states by the execution of the guarded commands, where a *state* is a unique valuation to program variables. The following is an intuitive explanation of the redesign algorithm presented in [19].

- *Step 1*: From every state $s$, identify the length of the shortest program computation that can reach a state where $\mathcal{T}$ has access to the shared resource. The length of such a shortest computation from $s$ is called the *rank* of $s$. Observe that the rank of a state where $\mathcal{T}$ has access to the shared resource is zero.
- *Step 2*: Identify the set of reachable states whose rank is infinity, denoted $\mathcal{S}_\infty$. That is, there is no program computation from any state in $\mathcal{S}_\infty$ that leads to a state where $\mathcal{T}$ has access to the shared resource.
- *Step 3*: Make the *trying states* that belong to $\mathcal{S}_\infty$ unreachable by eliminating the program transitions that terminate in a reachable state in $\mathcal{S}_\infty$. A *trying state* is a state in which the thread $\mathcal{T}$ is trying to gain access to the shared resource. This step may result in creating states that have no outgoing transitions, called the *deadlocked states*.
- *Step 4*: Make deadlock states unreachable using the transition elimination technique introduced in Step 3.
- *Step 5*: If an initial state becomes deadlocked, then declare that it is impossible to revise $p$ and terminate.
- *Step 6*: Update the ranking of all states as Steps 3 and 4 eliminate some program transition that may affect the length of the shortest computations to a state where $\mathcal{T}$ has access to the shared resource. Notice that the set of states $\mathcal{S}_\infty$ is also updated.
- *Step 7*: If there still is a reachable trying state with rank infinity, then go to Step 3. Otherwise, proceed to Step 8.
- *Step 8*: Eliminate any program transition that starts in a non-zero rank state and terminates in a state with a higher or equal rank. Such transitions participate in a livelock (i.e., non-progress cycle) that must be resolved so progress can be achieved.

The above algorithm determines whether or not there exists a revised version of the input reachability graph that meets the new progress property. While we express our algorithm in the context of a reachability graph of the input program, we have developed [18] a symbolic and distributed version of the above algorithm that analyzes each program action in isolation and determines the nature of the changes that should be made on each action so the entire program captures a new progress property.

After creating a *correct* synchronization skeleton in our high atomicity model, the abstract design is refined to a concrete design (e.g., message passing) under platform-specific constraints. For instance, for the refinement of high atomicity actions, we implement each high atomicity action as an atomic transaction on top of a Software Transactional Memory [25, 46]. Moreover, developers should be able to apply correctness-preserving refinement techniques that decrease the atomicity of our abstract synchronization skeletons. While several techniques [5] exists to support such correctness-preserving refinements, most of them rely on reasoning about parallel programs in the atomic model of concurrency [43], where it is assumed that no two operations are executed simultaneously. Such an assumption does not hold for multicore processors where multiple operations may be executed at the same physical moment in time. The development of efficient refinement algorithms is an open problem for our future work.

## 3 Example: Barrier Synchronization

In this section, we demonstrate our proposed approach for a Barrier Synchronization (BarSync) protocol that provides the underlying synchronization skeleton of numerous round-based parallel applications. In BarSync, $n$ different threads $T_i$ $(1 \leq i \leq n)$ concurrently execute in consecutive rounds, where in each round the execution of $T_i$ includes three steps of *initialization*, *execution* and *finalization*. Due to data dependencies, a thread can enter the next round only if all threads have finalized the previous round. We start with a straightforward design of BarSync that does not guarantee progress of threads. Then we apply our automated redesign algorithms (represented in the previous section) to automatically evolve the design of BarSync. The input design of BarSync is as follows:

$$\begin{array}{llll} A_{j1}: & pc_j = init & \longrightarrow & pc_j := execute; \\ A_{j2}: & pc_j = execute & \longrightarrow & pc_j := final; \\ A_{j3}: & pc_j = final & \longrightarrow & pc_j := init; \end{array}$$

In this section, we present a version of BarSync that includes three threads $T_1, T_2$, and $T_3$. Each thread $T_j$, $1 \leq j \leq 3$, has a variable $pc_j$ that represents the program counter of $T_j$. The program counter $pc_j$ can be in three positions init, execute and final. If thread $T_j$ is in the init position, then it changes its position to execute. From execute, the position of $T_j$ transitions to final, and then back to the init position. Let $\langle pc_1, pc_2, pc_3 \rangle$ denote a state of BarSync. The specification of BarSync requires that, starting from the state $allI = \langle$ init, init, init $\rangle$, all threads will eventually synchronize in the state $allF = \langle$ final, final, final $\rangle$ while at least two threads are always in the same position.

Observe that, in the input design of BarSync, one thread could circularly transition between the positions init, execute, final, thereby preventing other threads to finalize; i.e., averting the synchronization on $\langle$final, final, final$\rangle$. To ensure synchronization, we evolve the behaviors of the input BarSync by ensuring that starting from the state $\langle$init, init, init$\rangle$, the state $\langle$final, final, final$\rangle$ will eventually be reached. The below actions represent the revised design of $T_1$. The revised actions of $T_2$ and $T_3$ are structurally similar to those of $T_1$, hence omitted.

$$\begin{array}{ll} A'_{11}: & (pc_1 = init) \wedge \\ & (((\mathbf{pc_2} \neq \mathbf{final}) \wedge (\mathbf{pc_3} \neq \mathbf{final})) \vee \\ & ((\mathbf{pc_2} = \mathbf{final}) \wedge (\mathbf{pc_3} = \mathbf{final}))\ ) \\ & \qquad\qquad\qquad \longrightarrow \quad pc_1 := execute; \\ A'_{12}: & (pc_1 = execute) \wedge \\ & (\mathbf{pc2} \neq \mathbf{init}) \wedge (\mathbf{pc3} \neq \mathbf{init}) \\ & \qquad\qquad\qquad \longrightarrow \quad pc_1 := final; \\ A'_{13}: & (pc_1 = final) \wedge \\ & (((\mathbf{pc_2} = \mathbf{init}) \wedge (\mathbf{pc_3} = \mathbf{init})) \vee \\ & ((\mathbf{pc_2} = \mathbf{final}) \wedge (\mathbf{pc_3} = \mathbf{final}))\ ) \\ & \qquad\qquad\qquad \longrightarrow \quad pc_1 := init; \end{array}$$

From the init state, a thread can transition to the execute state if it is not the case that only one of the other two threads has finalized its computation in the current round. A thread can finalize from the execute state if none of the other two threads is in its initial state. A thread can start a new round of computation if either the other two threads have already transitioned to a new round or both have finalized the current round.

We would like to mention that, in addition to the BarSync presented in this section, we have automatically designed a version of BarSync with 18 threads using our distributed synthesis algorithm [18] on a cluster of 5 regular PCs (see [18] for our experimental results). Moreover, we have automatically redesigned the synchronization skeleton of several parallel/distributed programs such as mutual exclusion, readers/writers, agreement in the presence of faults and diffusing computation [19, 16]. In the next section, we investigate how existing parallel programming paradigms and software design methods address the issues of simplicity, generality, behavioral evolution and automated tool support.

## 4 Related Work

In this section, we discuss related work from two standpoints; models and paradigms for parallel programming and software engineering techniques for the design of concurrent systems.

### 4.1 Programming Models/Environments

We study explicit/raw threading (Pthreads and Win32 threads) [24], OpenMP [8, 37], Intel Threading Building Blocks (TBB) [44], Cilk [11], Microsoft Task Parallel Library (TPL) [35], and Message Passing Interface (MPI) [26], with respect to the simplicity of programming, generality of the computing model, support for behavioral evolution and tool support.

#### 4.1.1 Simplicity of Design/Programming

Explicit threading [24] provides a set of data types and function calls to enable multithreaded programming. While developers have full control over thread creation and synchronization, raw threading combines functional and low-level synchronization code, thereby increasing the complexity of parallel program design. Developing parallel programs in OpenMP [8, 37] is relatively easier compared with explicit threading in that programmers only specify the parts of their sequential code that can be executed in parallel, called *parallel regions*. Each parallel region has its own private variables and some shared variables. OpenMP simplifies some of the tasks such as thread creation/destruction, load balancing and loop parallelization. Intel TBB library [44] supports two main abstractions to simplify parallel programming, namely *tasks* and *templates*. Developers have to specify their parallel tasks instead of explicit thread creation; TBB creates, manages and destroys threads implicitly. Moreover, TBB provides templates for loop parallelization. Cilk [11] extends the C programming language in order to enable multithreading in the shared memory model. Cilk also includes a run-time system that provides a processor-oblivious view for parallel execution of threads, where threads are scheduled on the available processors in a transparent fashion. Microsoft Task Parallel Library (TPL) provides a set of abstractions in object-oriented programming for loop/task parallelization while leaving the design of synchronization for developers. The Message Passing Interface provides a library for programming over massively parallel/distributed machines, where processes communicate by exchanging messages either individually or in a group. The ready-to-use functions in the MPI library simplify the task of programming.

#### 4.1.2 Generality of Computing Model

In this section, we discuss the generality of each programming paradigm in terms of the abstractions it provides and its computation/communication model. The basic unit of abstraction in explicit threading is a thread of execution and the execution semantics is based on the non-deterministic interleaving of all threads. The program model is based on non-atomic instructions, and shared variables are used for inter-thread communication. The fork-join model of parallel computation is the core of some important programming environments such as OpenMP and Cilk. In the fork-join model a master thread creates a set of slave threads that compute in parallel, and upon finalization, the slave threads synchronize. The fork-join model is mostly appropriate for symmetric multithreading, where the synchronization skeleton of threads are similar; in some cases, the same piece of code is exactly replicated in separate threads and executed in parallel. As such, it is difficult to develop parallel applications in which threads may have different control structures.[2] In the task-based paradigms, tasks are the basic units of abstraction. For instance, in Intel TBB, independent units of computation are realized as parallel tasks with the support of a function library and a run-time system. The Microsoft TPL also supports a task-based model of parallel computation in object-oriented programming, where a class library provides the necessary data abstractions for task creation and synchronization.

#### 4.1.3 Behavioral Evolution (Change Management)

To the best of our knowledge, almost all the programming models/paradigms that we have considered in this section lack a systematic way for program redesign and behavioral evolution. While these approaches facilitate the *creation* of parallel application and the detection of behavioral errors, most of them lack a systematic/automatic method for reasoning about the global temporal behaviors (e.g., deadlock-freedom, data race avoidance and progress) in terms of local synchronization mechanisms in each thread. As such, developers have to manually get involved in (re)design of the synchronization skeleton of programs after detecting behavioral errors.

#### 4.1.4 Tool Support

The existing tool support for parallel programming paradigms is mostly focused on *detecting* behavioral errors of an implementation that includes both functional and synchronization code; less attention has been paid to automatic revision of synchronization skeletons. For example, there exist a variety of debugging tools for raw threading such as *Assure*, *Ladebug*, *TotalView*, Microsoft Visual Studio [4, 30], Cilk's *Nondeterminator* [2] and Intel Thread Checker [44] that mostly enable developers to (i) detect data races and deadlocks, and (ii) analyze performance. The *Allinea Distributed Debugging Tool* [3] provides support for detecting and visualizing behavioral errors in MPI programs. Additionally, several model checking tools [47, 23, 42] exist that verify whether or not parallel programs meet global temporal properties.

### 4.2 Software Design Methods

In this section, we briefly study four major software design techniques, namely Design-By-Contract (DBC) [40, 41], Aspect-Oriented Programming (AOP) [27, 28], the

[2]For simplicity, we selected a symmetric example in Section 3, nonetheless, our approach is general and we have automatically designed parallel programs with asymmetric threads [19, 16].

Universe Model (UM) [9, 10] and UNITY [12], with respect to simplicity of design in each approach, generality of design model in each method, support for behavioral evolution and tool support.

### 4.2.1 Simplicity of Design

In the DBC method [40, 41], each software component/module has a *contract* that specify its rights and responsibilities in interaction with other components/modules. More precisely, a component requires its environment to meet a *precondition* so it can provide a service. The component is responsible for (i) satisfying a *postcondition* if its precondition is satisfied, and (ii) honoring an *invariant* constraint while executing. For instance, in Systematic Concurrent Object-Oriented Programming (SCOOP) [41], developers need not be concerned about low-level concepts such as locks and semaphores for synchronization. Instead, they implement their system in the same way as sequential object-oriented programs. The SCOOP terminology provides a new modifier keyword, called separate, that means an object (respectively, a method) is executed on a separate processor. Inter-object synchronization is performed by argument passing and preconditions. That is, a routine r() can call a separate entity if that entity is an argument of r(). A call on a routine waits until all separate objects used in its precondition are available and its precondition holds.

The aspect-oriented methodology [27, 28] provides a method for design and implementation of the aspects that crosscut all components of a program. For example, in a multithreaded program, the aspect of synchronization is considered in all threads. To capture an aspect in an individual component, several pieces of code, called *advices*, are woven at specific places in the code of an exiting program, called the *joinpoints*. For concurrency, developers have to identify the joinpoints and the necessary code for synchronization in each thread. While AOP provides a powerful abstraction for modular implementation of crosscutting concerns, it is often difficult to identify the correct joinpoints and the advices that do not interfere with program's functional code.

The Universe Model (UM) [9, 10] provides a method for concurrent object-oriented programming, where an object can execute if it has exclusive access to all the objects it needs, called the *object universe* or the *object realm*. The universe of an object dynamically changes when an object is removed/added from/to its universe. Developers specify the needs and responsibilities of individual objects, called *contracts*, in a declarative language. The run-time system of the UM dynamically resolves the dependencies between the needs of objects to provide concurrency.

UNITY [12] provides a formal program logic along with a proof system for the design of parallel and distributed programs. A program comprises a set of variables and a set of guarded commands [14] that are executed non-deterministically. UNITY's program logic enables developers to first specify program behaviors and then use the rules of the proof system to refine specifications to programs in a stepwise fashion. While UNITY provides a fundamental theory for designing parallel programs, it is difficult for mainstream developers to design and reason about programs in UNITY due to the high level of formalism. By contrast, while our design language is similar to UNITY, we provide automated support for refinement and behavioral evolution.

### 4.2.2 Generality of Design Model

In DBC, the main unit of abstraction is a component along with its contracts, and designers should reason about program behaviors in terms of pre-postconditions. It is often the case that a run-time library (e.g., in SCOOPLI [41]) handles low-level concurrency issues such as deadlock prevention. As a result, DBC provides a straightforward and abstract model for concurrent computations, which is easy to learn and to refine to other models of computations such as raw multithreading.

In AOP, developers design programs by composing an aspect with a *base program* in a modular fashion. However, identifying the right joinpoints and the respective advices is difficult. For example, we have shown that adding fault tolerance aspects to parallel/distributed programs is a hard problem [32]. Since the AOP paradigm has been mostly realized in object-oriented languages, the basic unit of computation is often an object and the communication model is often based on method calls (e.g., AspectJ [28]). As such, aspect-oriented programs can easily be refined to message-passing parallel programs.

The basic unit of abstraction in the UM [9, 10] is the concept of the object universe in an object-oriented setting. While the UM is based on DBC, the contracts specified in the UM are in a higher level of abstraction in that they specify objects realms.

The basic units of abstraction in UNITY are specifications and programs. UNITY provides a fundamental change in mindset for the designers of parallel program as they have to think parallel from the outset. Moreover, UNITY programs are platform-independent and designers can refine programs further when new platform-specific constraints are introduced.

### 4.2.3 Behavioral Evolution (Change Management)

In DBC, given a new functionality, designers should determine what changes they should make in the pre-postconditions and the invariants of the existing components in order to capture the new functionality. Moreover, they should determine whether or not new components have to be introduced. Depending on the nature of the new functionality, this may be a difficult task. For example, if the new functionality is a new progress property that requires every component to eventually have access to some shared resource, then it is difficult to identify the local changes that should be made in each component and its contracts so the overall behaviors of the revised program captures the new progress requirement.

One of the most important problems targeted by AOP is the incremental addition of crosscutting functionalities. In this regard, AOP provides a method for architectural decomposition of distinct concerns across different components of a program. However, reasoning about the impact of different crosscutting concerns (e.g., fault tolerance and security) on each other remains a challenge. The program logic of UNITY [12] enables designers to formally specify new behaviors that should be added to an existing program. Moreover, the refinement rules of UNITY provide a systematic method for capturing new behaviors while preserving program correctness with respect to its specification.

#### 4.2.4 Tool Support

Most existing DBC tools/languages [39, 34, 33] that facilitate the design of robust programs rely on run-time assertion checking without enabling reasoning about the global properties of programs in terms of contracts. For instance, the Eiffel [39] programming environment supports DBC by providing necessary abstractions, diagrams, function libraries, a compiler and a debugger. The Java Modeling Language [34] augments Java with a interface specification language for contracts. Aspect-oriented design techniques [48] provide tool support for high-level design of programs in visual modeling languages. For example, Stein *et al.* [48] present a visual modeling language for specifying the design of aspect-oriented programs in the Unified Modeling Language (UML) [45]. To the best of our knowledge, the UM lacks a systematic approach for helping designers specify the changes in the universe of each object in such a way that a new global functionality is captured. The existing tool support for UNITY mostly relies on the use of theorem provers [20], which is difficult to use by mainstream developers.

## 5 Concluding Remarks and Future Work

We proposed a platform-independent framework for developing parallel applications, where more energy and resources are spent in design in order to potentially reduce development costs. Our framework comprises two layers, namely the *property-oriented* design layer and the *refinement* layer. The property-oriented design layer enables incremental evolution of the synchronization mechanisms of parallel programs upon detecting new global properties such as mutual exclusion, deadlock-freedom and progress (e.g., guaranteed service). We perform the property-oriented design in the shared memory model with a high atomicity program model, where threads can read/write program variables in an atomic step. In the refinement layer, we transform our high level design to more concrete design artifacts. Automating such correctness-preserving refinements is the subject of our future work. Compared with existing paradigms, our approach has a simpler and more general design language, and provides automated assistance for behavioral debugging.

While we have developed algorithms for the property-oriented design of fault tolerance, safety and progress properties [16, 19, 18], our future work will mostly be focused on (1) developing automated (re)design algorithms for a combination of global properties, (2) refining abstract synchronization skeletons for multicore systems, and (3) extending tool support for transforming our high-level design to common programming languages.

## References

[1] Resources on parallel patterns. http://www.cs.uiuc.edu/homes/snir/PPP/.

[2] *Cilk 5.4.6 Reference Manual*. Supercomputing Technology Group, MIT Laboratory for Computer Science, 1998.

[3] Allinea Software, 2008. http://www.allinea.com/.

[4] *MSDN, Visual Studio 2008 Developer Center*, 2008.

[5] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, 2006.

[7] M. Bakhouya, J. Gaber, and T. A. El-Ghazawi. Towards a complexity model for design and analysis of PGAS-Based algorithms. In *HPCC*, pages 672–682, 2007.

[8] B. Barney. *OpenMP*. Lawrence Livermore National Laboratory, April 2008.

[9] R. Behrends and K. Stirewalt. The universe model: an approach for improving the modularity and reliability of concurrent programs. In *ACM SIGSOFT Foundations of Software Engineering*, pages 20–29, 2000.

[10] R. Behrends, R. E. K. Stirewalt, and L. K. Dillon. A component-oriented model for the design of safe multithreaded applications. In *International Workshop on Component-Based Software Engineering*, pages 251–266, 2005.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

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

[13] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, 1993.

[14] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[15] M. Dubois and H. Lee. Stamp: A universal algorithmic model for next-generation multithreaded machines and systems. In *IPDPS*, pages 1–5, 2008.

[16] A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, May 2005.

[17] A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *IEEE/ACM international conference on Automated Software Engineering*, pages 44–53, 2007.

[18] A. Ebnenasir. Action-level addition of Leads-To properties to shared memory parallel programs. Technical Report CS-TR-08-01, Computer Science Department, Michigan Technological University, Houghton, Michigan, March 2008.

[19] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.

[20] S. O. Ehmety and L. C. Paulson. Mechanizing compositional reasoning for concurrent systems: some lessons. *Formal Aspects of Computing*, 17(1):58–68, 2005.

[21] E. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[22] A. H. Gebremedhin, M. Essaïdi, I. G. Lassous, J. Gustedt, and J. A. Telle. Pro: a model for the design and analysis of efficient and scalable parallel algorithms. *Nordic Journal of Computing*, 13(4):215–239, 2006.

[23] P. Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[24] M. Hays. Posix threads tutorial. `http://math.arizona.edu/~swig/documentation/pthreads`, December 2007.

[25] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

[26] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *NPC*, volume 5245, pages 266–275, 2008.

[27] G. Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[29] S. Konrad and B. H. C. Cheng. Automated analysis of natural language properties for uml models. In *MoDELS Satellite Events*, pages 48–57, 2005.

[30] B. Kuhn, P. Petersen, and E. O'Toole. OpenMP versus threading in c/c++. *Concurrency - Practice and Experience*, 12(12):1165–1176, 2000.

[31] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 441–449, 2003.

[32] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, July-September 2005.

[33] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[34] G. T. Leavens, C. Ruby, K. Rustan, M. Leino, E. Poll, and B. Jacobs. JML (poster session): notations and tools supporting detailed design in java. In *OOPSLA'00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, 2000.

[35] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, October 2007.

[36] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 61, 1995.

[37] N. Matloff. Introduction to openmp. Technical report, Department of Computer Science, University of California at Davis, 2008.

[38] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[39] B. Meyer. Eiffel: a language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[40] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[41] B. Meyer. Systematic Concurrent Object-Oriented Programming (SCOOP). *Communications of the ACM*, 36(9):56–80, 1993.

[42] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 362–371, 2008.

[43] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[44] J. Reinders. *Intel Threading Building Blocks, Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, first edition, 2007.

[45] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[46] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[47] S. F. Siegel and G. S. Avrunin. Verification of halting properties for MPI programs using nonblocking operations. In *14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 326–334, 2007.

[48] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In *AOSD'02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, 2002.