Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs

Xianghua Deng, Matthew B. Dwyer, John Hatcliff and Masaaki Mizuno
Kansas State University
Department of Computing and Information Sciences
Manhattan, KS 66506, USA
{deng,dwyer,hatcliff,masaaki}@cis.ksu.edu

ABSTRACT

Concurrency is used in modern software systems as a means of addressing performance, availability, and reliability requirements. The collaboration of multiple independently executing components is fundamental to meeting such requirements and such collaboration is realized by synchronizing component execution.

Using current technologies developers are faced with a tension between correct synchronization and performance. Developers can be confident when simple forms of synchronization are used, for example, locking all accesses to shared data. Unfortunately, such simple approaches can result in significant run-time overhead, and, in fact, there are many cases in which such simple approaches cannot implement required synchronization policies. Implementing more sophisticated (and less constraining) synchronization policies may improve run-time performance and satisfy synchronization requirements, but fundamental difficulties in reasoning about concurrency make it difficult to assess their correctness.

This paper describes an approach to automatically synthesizing complex synchronization implementations from formal high-level specifications. Moreover, the generated coded is designed to be processed easily by software model-checking tools such as Bandera. This enables the generated synchronization solutions to be verified for important system correctness properties. We believe this is an effective approach because the tool-support provided makes it simple to use, it has a solid semantic foundation, it is language independent, and we have demonstrated that it is powerful enough to solve numerous challenging synchronization problems.

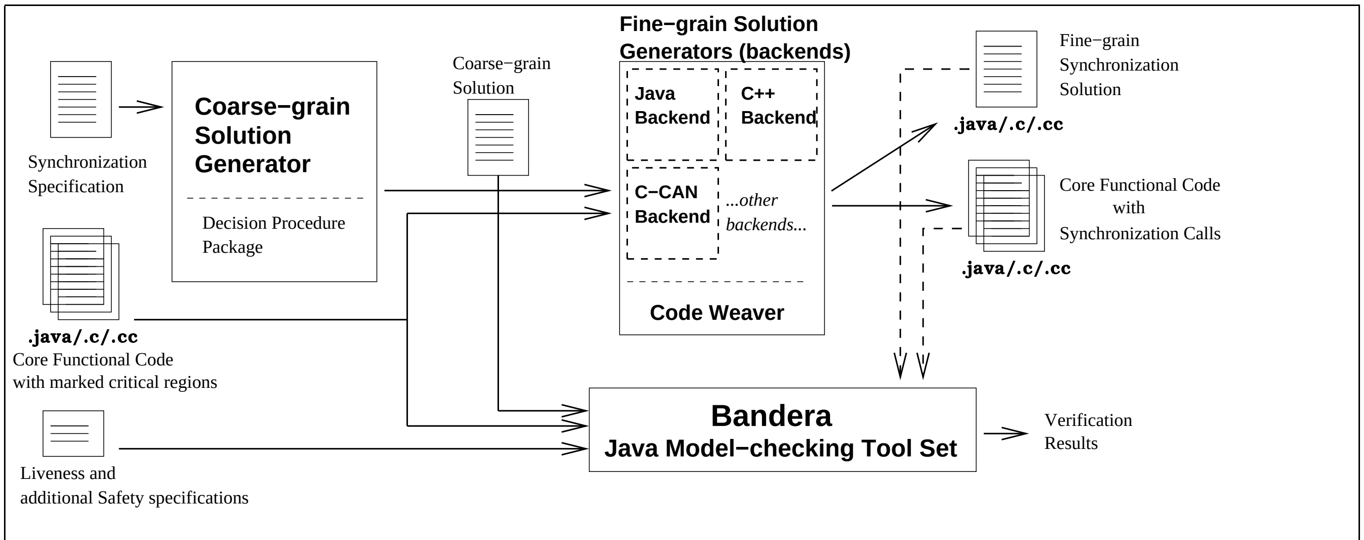
1. INTRODUCTION

The use of concurrency to resolve challenging system requirements with software is becoming common-place. Concurrency can, for example, enable increased system performance through the use of parallel processing, enable increased availability and reliability through replication of functionality, and, through pre-emptive execution, enable timely system response. Increasingly, developers of concurrent systems are reusing existing libraries and programming frameworks rather than developing entire applications from scratch. One of the key challenges in this setting is enforcing application specific synchronization policies on the mixture of reused and custom-built components that comprise the application.

Synchronization can be viewed as controlling the possible scheduling of component execution. For all but the simplest of systems, the number of ways that components can be scheduled is enormous. This, combined with the fact that synchronization is distributed throughout the application, makes it very difficult to reason about the overall correctness of an application with respect to synchronization requirements. This presents developers of concurrent systems with a choice between employing the synchronization policies that are best suited to their application, but which may be difficult to validate, or employing very simple policies that are overly restrictive of component scheduling, but which are easy to validate. Given the lack of usable and effective methodologies for introducing complex synchronization policies into software, developers invariably choose to use simple synchronization policies even at the expense of system performance.

In this paper, we describe an approach which addresses these issues by: (*i*) allowing developers to specify global synchronization policies that govern the execution of components in a concurrent application; (*ii*) automatically synthesizing correct efficient implementations of those synchronization policies; and (*iii*) independently verifying the correctness of those implementations and supporting the automated checking of other system correctness properties. Ours is a focused approach that does not attempt to assure complete functional correctness. We believe that, except for the most safety critical systems, formal specification and verification of complete functional behavior is impractical. We also feel that generation of code from high-level specifications is impractical for complete functional behavior. However, we believe that formal definition and derivation of code for particular *aspects* of program behavior is practical, and in this paper we give what we believe is a convincing demonstration of this for the aspect of synchronization.

In our approach (refer to Figure 1), the developer builds core functional code without implementing the synchronization policy. For example, in a readers-writers system, code to access a shared buffer would be written. The developer delimits particular code regions that will be synchronized. In a readers-writers system, one would mark the regions that access shared data as a reader and those that access the data as a writer. A synchronization policy that controls the scheduling of threads that attempt to execute in those regions is specified in a language of global invariant patterns. This synchronization specification is compiled to a predicate representing an invariant that must hold on the regions. These invariants can then be used to generate implementations of the synchronization policy in Java, for example, that are then integrated with the



Synchronization Specification
Coarse−grain Solution Generator
Decision Procedure Package
.java/.c/.cc
Core Functional Code with marked critical regions
Liveness and additional Safety specifications
Coarse−grain Solution
Fine−grain Solution Generators (backends)
Java Backend
C++ Backend
C−CAN Backend
...other backends...
Code Weaver
Fine−grain Synchronization Solution
.java/.c/.cc
Core Functional Code with Synchronization Calls
.java/.c/.cc
Bandera Java Model−checking Tool Set
Verification Results

Figure 1: SyncGen Tool Architecture

core functional code to produce a complete concurrent application.

This process guarantees that the safety properties described by the invariant are correctly implemented by construction. To guard against errors in the synthesis process we can encode specifications of the intended synchronization properties into the generated code and apply model checking to verify that they hold. This kind of checkable redundancy yields a high degree of confidence in the correctness of our synthesis approach. This confidence does not, however, extend to additional correctness properties, e.g., liveness properties, that one might wish to check. To address this, we have adapted the Bandera [8] toolset to extract models that are a *hybrid* of synchronization specifications and core functional code. The resulting models encode program synchronization very efficiently, yielding dramatic state-space reductions that promise to increase the size and complexity of systems for which model checking is tractable.

Our approach builds off and extends the global invariant approached advocated by Andrews [2]. Our main extensions have been to identify recurring patterns of global invariant specification and to revise the methodology for developing implementations of those specifications [22]. In this paper, we further extend the work by providing automated tool support for the synthesis of implementations. We believe the result is a framework for developing correct efficient synchronization code that is:

- *Powerful* – using the pattern system, complicated structures can be described clearly and succinctly at a very high level;
- *Expressive* – we have used the pattern system to specify solutions for a wide variety of exercises from three well-known concurrency texts [2, 3, 13];
- *Automatic* – this is push-button approach where code with very intricate semantics is automatically generated from high-level specifications;
- *General* – the approach is language independent and supports multiple languages and synchronization primitives;
- *Formal* – our aspect specification language has a rigorous semantic foundation which enables code generation techniques that use decision procedures for a quantifier-free fragment of first-order arithmetic and yields a high degree of confidence in the generated solutions; and
- *Verifiable* – the structure of the generated code and associated artifacts is designed so that crucial correctness requirements can be checked automatically using existing software model-checking technology.

The SyncGen tool described in this paper has been fully implemented[1]. While this paper focuses on the synthesis and verification of synchronization in Java programs, in collaboration with colleagues we have produced synchronization code generators for other languages and execution platforms. These include C++ with POSIX shared memory primitives, distributed synchronization primitives in C on CAN networks [24], and most recently we have begun adaptation of the event-channel mechanism in TAO [25], a widely used object request broker, to implement synchronization policies in real-time distributed object-based systems.

The rest of the paper is organized as follows. Section 2 describes the semantic foundation of global invariants, how they give rise to synchronization solutions, and how common global invariants can be captured by a collection of reusable patterns. Section 3 describes the generation of coarse-grain solutions and how these solutions can be optimized for specific patterns. Section 4 summarizes the construction of a fine-grain solution for Java. Section 5 presents a non-trivial example that involves the composition of multiple primitive synchronization policies in a single application. Section 6 describes our approach to verifying the correctness of synchronization implementations and how we can leverage the coarse-grain solution for state-space reduction in model checking. Section 7 describes related work and Section 8 concludes.

2. SPECIFYING SYNCHRONIZATION

2.1 Global invariants

In our approach to coding concurrent software, a developer uses traditional methods and development environments to produce what we term the system's *core functional code*. This code realizes the behavior of each concurrent component of the system, but *does not* specify how components synchronize. Instead, the developer simply marks the *regions* of code in each component that require synchronization with syntactic tags. The developer then partitions regions into equivalence classes termed *clusters*. Intuitively, regions R_1 and R_2 should be in the same cluster if a thread at region R_1

[1]The tool and a collection of examples are publicly available at `www.cis.ksu.edu/saves`.

waits for an event or state that is triggered or changed by a thread at R_2.

Each region R is associated with two *conceptual counters* In_R and Out_R that keep track of the number of times that a thread of control has entered or exited the region, respectively. The developer then specifies the synchronization policy for each cluster by giving a logic formula that constrains the relationships between the cluster's region counters.

As an example, consider how a developer would specify a simple system of concurrent readers and writers of a shared variable. Figure 2 displays the core functional code for the `Reader` and `Writer` threads. In this code, the regions that need synchronization are the reading and writing of the shared variable. These regions form a cluster since a thread entering the reader region must wait for a thread in the writer region to exit the region (and vice versa) if the proper mutual exclusion discipline is to be enforced. Thus, the developer tags the entrance and exit of the regions with special comments recognized by the SyncGen tool. These regions should be synchronized so that (a) multiple reader threads can be in the reader region provided no writer thread is in the writer region, and (b) one writer thread is in the writer region provided no reader thread is in the reader region. The logic formula that specifies this policy is

$$\begin{aligned} I_{rw} \overset{\text{def}}{=} \quad & (\mathit{In}_R - \mathit{Out}_R = 0 \lor \mathit{In}_W - \mathit{Out}_W = 0) \\ & \land (\mathit{In}_W - \mathit{Out}_W \le 1) \end{aligned}$$

where the variables used are the implicit entrance and exit counters associated with each region. The first line specifies that either the reader region is empty or the writer region is empty; the second line specifies that at most one thread can be in the writer region at a time.

2.2 Global invariant patterns

One possible drawback of the global invariant approach is that developers may find it difficult to identify appropriate global invariant formulas that accurately capture the safety properties of the given synchronization requirements. To address this problem, we provide a set of global invariant *patterns* or idioms that can be used directly or composed to produce more complex synchronization specifications [23]. Below we describe each pattern and present its formal semantics as a global invariant formula.

Bound(R, n): at most n threads can be in R at any point in time. The underlying formalization is $(\mathit{In}_R - \mathit{Out}_R \le n)$.

Exclusion$(R_1, R_2, \cdots, R_n)$: at any point in time, threads can be in at most one synchronization region out of the n synchronization regions. To aid in the formalization, we define a predicate *OnlyOneOccupied*(i,n) that holds in the state in which threads are only in one region R_i out of n regions:

$$\begin{array}{lll} & (\mathit{In}_1 - \mathit{Out}_1 = 0) & \land \quad \cdots \\ \land & (\mathit{In}_{i-1} - \mathit{Out}_{i-1} = 0) & \land \quad (\mathit{In}_{i+1} - \mathit{Out}_{i+1} = 0) \\ \land & \cdots & \land \quad (\mathit{In}_n - \mathit{Out}_n = 0). \end{array}$$

Thus, the formalization is $\lor_{i \in \{1, \ldots, n\}} \mathit{OnlyOneOccupied}(i, n)$. For example, *Exclusion*(R_1, R_2, R_3) is

$$\begin{array}{ll} & (\mathit{In}_1 - \mathit{Out}_1 = 0) \land (\mathit{In}_2 - \mathit{Out}_2 = 0)) \\ \lor & (\mathit{In}_1 - \mathit{Out}_1 = 0) \land (\mathit{In}_3 - \mathit{Out}_3 = 0)) \\ \lor & (\mathit{In}_2 - \mathit{Out}_2 = 0) \land (\mathit{In}_2 - \mathit{Out}_2 = 0)). \end{array}$$

Resource$((R_P, N_P), (R_C, N_C), n)$: a pool of resources (with n items in the pool initially) accessed by a producer region R_P that produces N_P resource items each time a thread executes it, and a consumer region R_C that consumes N_C resource items each time a thread executes it. If there are fewer than N_C resource items in the pool, a thread executing R_C waits at the entrance of the region until there are at least N_C resources in the pool. Thus, the formalization is $(\mathit{In}_C \le ((\mathit{Out}_P * N_P + n)\ \mathit{div}\ N_C)$ where *div* represents integer division.

Barrier(R_1, R_2): the k^{th} thread to enter R_1 and the k^{th} thread to enter R_2 meet at their respective synchronization regions and leave together. The formalization is $(\mathit{Out}_1 \le \mathit{In}_2) \land (\mathit{Out}_2 \le \mathit{In}_1)$.

The barrier pattern yields a symmetric synchronization; threads cannot move faster through R_1 than they can through R_2 (or vice versa). This pattern may be decomposed and its components used to specify an asymmetric *relay* synchronization described below.

Relay(R_1, R_2): a thread entering R_1 can leave R_1 immediately; however, the k^{th} thread entering R_2 is blocked and cannot leave R_2 until the k^{th} thread arrives at R_1. In this situation, an arrival of a thread at R_1 triggers a release of a thread at R_2. The formalization is $\mathit{Out}_2 \le \mathit{In}_1$.

It is convenient to extend the barrier synchronization to the following more general form, called the *Group* synchronization.

Group$((R_1, N_1), (R_2, N_2), \cdots, (R_n, N_n))$: N_i threads entering R_i for $1 \le i \le n$ meet, form a group, and leave the respective synchronization regions together. For example, let $n = 3$, $N_1 = 2$, $N_2 = 3$, and $N_3 = 4$. Then, 2 threads in R_1, 3 threads in R_2, and 4 threads in R_3 form a group and leave together. The formalization is

$$\land_{\,i \in \{1, \ldots, n\}} \ \land_{\,j \in \{1, \ldots, n\}} (\mathit{Out}_i \le (\mathit{In}_j\ \mathit{div}\ N_j) * N_i)).$$

The constraint $\mathit{In}_j\ \mathit{div}\ N_j$ gives the number of complete units that have entered region R_j. Now, multiplying $(\mathit{In}_j\ \mathit{div}\ N_j)$ by N_i and using this as a bound for Out_i for each R_j ensures that every time a complete unit assembles at each R_j for all j, another unit (N_i threads) is allowed to leave R_i. Thus, the k^{th} unit of threads in R_i may leave the region when the k^{th} unit of threads have entered every region R_j.

Pattern-based synchronization specifications are formed by composing instances of the above patterns where composition is interpreted as logical conjunction. For example, the second section of Figure 2 displays readers/writers synchronization specification expressed as a composition of *Exclusion* and *Bound* patterns. In this example, the developer declares a name `RW` for the single cluster, names `Reader`, `Writer` for the two synchronization regions, and finally the global invariant for the cluster. In problems with multiple clusters, this information is repeated for each cluster. Expanding the patterns to logical predicates as described above and interpreting the composition symbol '`+`' as conjunction yields the global invariant I_{rw}.

3. COARSE-GRAIN SOLUTION

A global invariant specification is automatically translated into an implementation independent *coarse-grain* synchronization solution which is represented using atomic test-and-update constructs $\langle \textbf{await}\ B \rightarrow S \rangle$. Informally, a thread executing $\langle \textbf{await}\ B \rightarrow S \rangle$ is suspended until boolean expression B becomes *true*. Once B holds, the thread continues in an atomic step to execute S — a sequence of one or more assignment statements.

An **await** statement will be placed at the entrance and exit of each region. Each expression B will act as a guard ensuring that threads can only enter/exit a region when an appropriate condition on region counter variables is satisfied, and S will be an increment of the appropriate region entrance/exit counter variable. Often, a

Core Functional Code With Delimited Regions:

```
final class Reader extends Thread {
 public void run() {
   while (true) {
   ...other computation...
   /*** RW Reader enter ***/
   ...read value of shared variable...
   /*** RW Reader exit  ***/
   ...other computation...
   }}}
```

```
final class Writer extends Thread {
 public void run() {
   while (true) {
   ...other computation...
   /*** RW Writer enter ***/
   ...write value to shared variable...
   /*** RW Writer exit ***/
   ...other computation...
   }}}
```

Synchronization Specification:

```
CLUSTER: RW;
REGIONS: Reader, Writer;
INVARIANT: Exclusion(Reader,Writer) + Bound(Writer,1);
```

Coarse-grain Solution:

```
CLUSTER: RW

REGION: Reader
ENTER: <AWAIT Writer_in - Writer_out == 0
          --> Reader_in++>
NOTIFY: ;
NOTIFYALL: ;
EXIT: <Reader_out++>
NOTIFY: ;
NOTIFYALL: Writer_in;
```

```
REGION: Writer
ENTER: <AWAIT Reader_in - Reader_out == 0  &&
              ((Writer_in + 1) - Writer_out) <= 1
          --> Writer_in++>
NOTIFY: ;
NOTIFYALL: ;
EXIT: <Writer_out++>
NOTIFY: Writer_in;
NOTIFYALL: Reader_in;
```

Woven Core Functional Code:

```
final class Reader extends Thread {
 public void run() {
   while (true) {
   ...other computation...
   ///*** RW Reader enter ***/
   SGCluster$RW.Reader$enter();
   ...read value of shared variable...
   ///*** RW Reader exit  ***/
   SGCluster$RW.Reader$exit();
   ...other computation...
   }}}
```

```
final class Writer extends Thread {
 public void run() {
   while (true) {
   ...other computation...
   ///*** RW Writer enter ***/
   SGCluster$RW.Writer$enter();
   ...write value to shared variable...
   ///*** RW Writer exit ***/
   SGCluster$RW.Writer$exit();
   ...other computation...
   }}}
```

Figure 2: SyncGen Tool Artifacts for Readers/Writers Example

guard B in $\langle$**await** $B \rightarrow S\,\rangle$ will be the constant **true** (signifying that a thread can unconditionally enter/exit a region); we abbreviate such **await** statements as $\langle\, S\,\rangle$ and refer to them as *atomic* statements.

In summary, our strategy ensures (via induction on the number of execution steps) that an invariant holds throughout an execution by (a) checking that the invariant holds in the initial state where all region counter variables have a value of 0 (base case), and (b) constructing appropriate guards at region boundaries (entrances/exits) that guarantee that, if the invariant holds before a thread crosses a boundary, then it will also hold after the thread crosses a boundary (induction case).

Figure 2 (middle) presents the coarse-grain solution for the readers/writers example that is produced automatically by the SyncGen tool. The solution passes along the declared cluster and region names, gives the await/atomic statement associated with each region entrance/exit, and specifies other regions that should be notified upon completion of each region entrance/exit. The intuition behind the guards for each region entrance is as follows: the reader region can only be entered when there are no threads present in the writer region (*i.e.*, when the difference between the number of threads that have entered the writers region and the number of threads that have exited the writers region is zero), and the writer region can only be entered when (a) there are no threads present in the reader region and (b) there are no threads present in the writer region. It is clear that more compact renderings of condition (b) are possible, but the displayed version is what produced by the automatic construction outlined in Section 3.1 below.

The SyncGen tool actually provides two different mechanisms for generating coarse-grain solutions. The first is an approach that starts from the logic formula representation of the global invariant (e.g., the formula I_{rw}) and employs weakest-precondition calculations and subsequent reductions using a decision procedure for a subset of first-order logic (our implementation uses the Stanford Validity Checker (SVC) decision procedure package). The second mechanism generates the coarse-grain representation directly from the global invariant pattern specification. Working directly from the patterns makes generation easier in several respects because information about the structure of the synchronization solution is already coded in the pattern concept – the structure does not have to be (re)discovered by manipulation of the invariant formula. We almost always use the pattern-based generation mechanism since (a) the generation process is more efficient — the formula-based method requires numerous calls to the decision procedure package,

(b) the generated solution is slightly more efficient due to leveraging structural information in the patterns, and (c) the pattern collection is expressive enough to specify a wide variety of synchronization solutions. The formula-based mechanism can be used when a particular global invariant cannot be expressed using the existing pattern collection. We give an overview of both approaches below. We give a more detailed explanation of the formula-based approach since it provides the semantic foundation of the coarse-grain solution generation.

3.1 Formula-based generation

Establishing the base case of our inductive argument is straightforward: references to region counter variables in the global invariant I are replaced with their initial value 0, and the decision procedure is called to verify that the resulting formula is true. If the formula is not true the user is notified that the proposed synchronization policy is unsatisfiable; otherwise, we continue with the inductive steps as described below.

To see how each entrance $\langle \textbf{await}\, B \;\rightarrow\; \mathit{In}_R\texttt{++} \rangle$ statement is constructed automatically from a given global invariant formula I, note that, in order to preserve I, a thread at the entrance of a region R must wait until it is guaranteed that incrementing In_R will preserve I (similarly for the exit of R). Thus, we are looking for the least restrictive condition B that guarantees that In_R++ will result in I being true. In other words, calculate the least restrictive B such that the Hoare triple $\{I \wedge B\}\, \mathit{In}_R\texttt{++}\, \{I\}$ is satisfied. Note that if all region guards are constructed appropriately, I should always be true, and thus it appears in both the precondition and postcondition of the triple.

Step 1: The calculation process begins by noting that a correct but often unnecessarily complex B_0 can be produced by taking $B_0 \overset{\text{def}}{=} \text{wp}(\mathit{In}_R\texttt{++}, I)$, i.e., B_0 is the weakest precondition of In_R++ with respect to I. Calculating $\text{wp}(S, I)$ when S is an assignment statement $x := e$ proceeds by substituting e for any occurrences of x in I: $\text{wp}(x := e, I) \overset{\text{def}}{=} I[e/x]$ [12]. In the readers/writers example, to calculate the **await** statement for the entrance of the reader region, we have

$$\begin{array}{lll} B_0 & \overset{\text{def}}{=} & \text{wp}(\mathit{In}_R := \mathit{In}_R + 1, I_{rw}) \\ & = & \quad (\mathit{In}_R{+}1 - \mathit{Out}_R = 0 \vee \mathit{In}_W - \mathit{Out}_W = 0) \\ & \wedge & \quad (\mathit{In}_W - \mathit{Out}_W \leq 1). \end{array}$$

Notice that the structure of this formula is more complex than necessary. For example, the first disjunct can never hold because of two *basic properties* of region counters: (1) counter variables can never have negative values, and (b) for any region R_i, $\mathit{In}_i \geq \mathit{Out}_i$. In addition, since I_{rw} holds globally, the second conjunct must be true since it also appears as a top-level conjunct in I_{rw}. Reducing B_0 based on these two observations yields $\mathit{In}_W - \mathit{Out}_W = 0$ — the guard at the reader region entrance (Figure 2). The following two steps give a method for carrying out such reductions for region entrances in a systematic way regardless of the structure of I.

Step 2: Let B_1 be the disjunctive normal form of B_0, that is, B_1 has form $C_1 \vee C_2 \vee \ldots \vee C_n$. Compute a smaller guard B_2 by removing disjuncts C_i for which SVC can establish that $I \wedge A \Rightarrow \neg C_i$ holds. Here, $\Rightarrow$ is implication, and A is a formula encoding the basic properties of counters mentioned above (e.g., it is a conjunction of facts such as $\mathit{In}_i \geq 0$, $\mathit{Out}_i \geq 0$, $\mathit{In}_i \geq \mathit{Out}_i$, etc.)

Picking up the readers/writers example, the disjunctive normal form of B_0 is

$$\begin{array}{ll} B_1 \overset{\text{def}}{=} & (\mathit{In}_R{+}1 - \mathit{Out}_R = 0) \wedge (\mathit{In}_W - \mathit{Out}_W \leq 1) \\ & \vee\, (\mathit{In}_W - \mathit{Out}_W = 0) \quad \wedge (\mathit{In}_W - \mathit{Out}_W \leq 1). \end{array}$$

Let C_1, C_2 represent the first and second disjuncts above, respectively. SVC can prove $I \wedge A \Rightarrow \neg C_1$ using facts encoded in A as described above. However, SVC fails to prove $I \wedge A \Rightarrow \neg C_2$ and so we have the reduced formula

$$B_2 \overset{\text{def}}{=} (\mathit{In}_W - \mathit{Out}_W = 0) \wedge (\mathit{In}_W - \mathit{Out}_W \leq 1).$$

Let D_1, D_2 be the first and second conjuncts in B_2.

Step 3: Compute a smaller guard B_3 from B_2 by removing conjuncts D_k from each of the remaining C_j in B_2 if $I \wedge A \Rightarrow D_k$ or D_k is entailed by $I \wedge A$ conjoined with other conjuncts in C_j.

In the example B_2 above, the decision procedure can establish that $I_{rw} \wedge A \Rightarrow D_2$ (intuitively, because D_2 appears as a top-level conjunct in I_{rw}). In fact, because the situation where a top-level conjunct in I_{rw} matches a D_i occurs so often, our tool makes a syntactic check for such cases before invoking the decision procedure processing. In the readers/writers example, the resulting guard for the entrance of the reader region is

$$B_3 \overset{\text{def}}{=} (\mathit{In}_W - \mathit{Out}_W = 0)$$

and this matches the reader guard displayed in Figure 2. The generation of the await statement in Figure 2 for writer region entrance follows the same steps.

Calculation of region exit statements begins with the weakest-precondition calculation of **Step 1**. Reduction then proceeds as in **Step 2** and **Step 3**, but instead of reducing based on information represented by $I \wedge A$, we reduce based on a formula $I \wedge A \wedge A'$ where A' represents additional facts that hold when a thread is inside the current region – this information is derived by calculating the strongest post-condition of the previously generated entrance statement. For example, here is the intuition for the reader exit: calculating $\text{wp}(\mathit{Out}_R := \mathit{Out}_R + 1, I_{rw})$ and converting to disjunctive normal form yields

$$\begin{array}{ll} B_1 \overset{\text{def}}{=} & (\mathit{In}_R - (\mathit{Out}_R{+}1) = 0) \wedge (\mathit{In}_W - \mathit{Out}_W \leq 1) \\ & \vee\, (\mathit{In}_W - \mathit{Out}_W = 0) \qquad \wedge (\mathit{In}_W - \mathit{Out}_W \leq 1). \end{array}$$

Unlike the calculation for entrance, **Step 2** does not lead to reductions because it is possible for each of disjuncts C_1, C_2 above to hold (i.e., checking $I \wedge A \Rightarrow \neg C_i$ fails). However, in **Step 3**, each of the occurrences of $(\mathit{In}_W - \mathit{Out}_W \leq 1)$ can be removed as in the case for the reader entrance (this follows from I_{rw}). Finally, for I_{rw} to hold while a thread is inside the reader region, it must be the case that $\mathit{In}_W - \mathit{Out}_W = 0$. This causes the potential guard B_1 to reduce to *true*, which yields an atomic statement for the reader exit.

Figure 2 illustrates that the coarse-grain solution also contains *notification information*. When an entrance/exit counter for region R is incremented, it may cause an entrance/exit guard of another region R' that was previously false to become true. Fine-grain solutions often implement await statements by blocking on false guards. In such implementations, a thread causing the guard at the entrance/exit of region R' to change from false to true should wake up threads waiting at the entrance/exit of R'. The `NOTIFY` clauses in Figure 2 indicate situations where one thread in a region entrance/exit should be awakened, and `NOTIFYALL` clauses indicate situations more than one thread in a region entrance/exit should be awakened. Note that notifying all threads yields a less efficient but guaranteed safe solution. Distinct `NOTIFY` and `NOTIFYALL` lists are maintained because issuing a `notify` is typically more efficient that issuing a `notifyall`. Although this information isn't necessary to specify the coarse-grain semantics, we include it at the coarse-grain level because (a) most fine-grain backends make use of it, thus this functionality is factored out of the back-ends, and (b) it is very easy to generate precise notification information using the pattern-based method described below – deferring the calculation to the back-ends where the structural information contained in the patterns has already been compiled away usually yields more conservative and thus less efficient notification actions.

3.2 Pattern-based generation

The pattern-based generation of region boundary guards leverages three facts: (1) region enter/exits have a regular form – a single counter increment, (2) patterns are *composed by conjunction* to form a synchronization specification, and (3) *weakest-precondition distributes across conjunction* [12], i.e.,

$$\text{wp}(S, I_1 \wedge \ldots \wedge I_n) = \text{wp}(S, I_1) \wedge \ldots \wedge \text{wp}(S, I_n).$$

Consider S to be a region counter increment and I_j to be the formulas obtained by instantiating j global invariant patterns (for $1 \leq j \leq n$). The law of distributivity above allows one to precompute entrance/exit *guard schemas* for each pattern individually. Then, given a pattern-based synchronization specification that contains pattern instances $P_1, \ldots, P_n$, a guard G for a region entrance/exit is built by conjoining guards G_j that are produced by instantiating the precomputed guard schemas for each P_j. Each G_j corresponds to a reduced $\text{wp}(S, I_j)$. We will illustrate this by considering several patterns below – details for the remaining patterns and correctness proofs can be found in [9].

Bound(R, n): Here, we have $I = (\mathit{In}_R - \mathit{Out}_R \leq n)$. The entrance guard schema for region R is

$$\text{wp}(\mathit{In}_R\text{++}, I) = ((\mathit{In}_R + 1) - \mathit{Out}_R \leq n).$$

The entrance guard schema for any region $R' \neq R$ is *true* (i.e., no guard condition is required to ensure that I holds after an entrance to R') since I holds before $\mathit{In}_{R'}$++ and therefore it must be true after $\mathit{In}_{R'}$++.

The nonreduced exit guard schema for region R is

$$\text{wp}(\mathit{Out}_R\text{++}, I) = (\mathit{In}_R - (\mathit{Out}_R + 1) \leq n).$$

However, since we know I must hold before Out_R++ and counters are always non-negative, the proposed exit guard will also be satisfied. Therefore the reduced exit guard for region R is *true*. Reasoning similar to the entrance guard for region $R' \neq R$ establishes that the exit guard schema for R' with respect to *Bound*(R, n) is *true*.

For generating notification information specific to this pattern, note that the only non-trivial guard is the entrance guard calculated above, and only the increment of Out_R at the region R exit can cause the guard to change state from *false* to *true*. Thus, upon exit of R, threads at the entrance to R should be notified. Note that `notify` should be used instead of `notifyall`: if a thread has been waiting at the entrance of a "full" R (i.e., n are currently in R), then one thread exiting will only allow one more thread to enter. Thus, only one thread (instead of all) should be woken up.

Exclusion(R_1, R_2): We consider a simplified version of exclusion; it is easy to see how to scale the steps to the n-way version of Section 2. Here we compute four guard and notification schemas: R_1 entrance and exit and R_2 entrance and exit. Working in a fashion similar to the *Bound* case above, we have

$$\begin{array}{rcl} G_1^{in} & = & (\mathit{In}_2 - \mathit{Out}_2 = 0) \\ G_1^{out} & = & \mathit{true} \\ G_2^{in} & = & (\mathit{In}_1 - \mathit{Out}_1 = 0) \\ G_2^{out} & = & \mathit{true}. \end{array}$$

For notification information, we have R_1 exit calls `notifyAll` on R_2 entrance, and R_2 exit calls `notifyAll` on R_1 entrance.

Considering the readers/writers specification of Figure 2, the distributivity of 'wp' allows us to assemble the course-grain solution from the schemas above as follows. For the reader entrance the *exclusion* pattern contributes $\mathit{In}_W - \mathit{Out}_W = 0$ but the *bound* pattern's contribution is trivial since we noted above that for all regions R' other than the region R that is bounded, the entrance guard is *true*. For the reader exit, both the *exclusion* and *bound* yield *true* guards so an atomic statement is generated (similarly for the writer exit). For the writer entrance the *exclusion* pattern contributes $\mathit{In}_R - \mathit{Out}_R = 0$ and the *bound* pattern contributes $(\mathit{In}_W + 1) - \mathit{Out}_W \leq 1$. Thus, the conjunction of these two forms the guard. The schemas above reveal that notification only occurs at the region exits. For the reader exit, *exclusion* contributes `notifyAll` to the writer entrance; *bound* contributes nothing. For the writer exit, *exclusion* contributes `notifyAll` to the reader entrance; *bound* contributes `notify` to the writer entrance.

4. FINE-GRAIN SOLUTION

As noted in the introduction, the language-independent coarse-grain solution skeleton can be translated to fine-grain solutions rendered using a variety of languages and synchronization primitives. In this section, we focus on the Java translation.

The translation to Java involves generating several methods and locks objects for each **await** and each atomic statement. Also, the counter variables and associated increments for each critical region must be implemented so as to ensure exclusive access across a cluster. All such definitions for a particular cluster are collected into common static class `SGCluster$`*clname*, and we outline the construction of each of these below. The translation follows Mizuno's strategy of implementing *specific notification*, and further motivation for this implementation is given in [21].

Counters: For each region *rname*, declare private static integer variable implementing the region's entrance/exit counters:

```
private static int <rname>_in, <rname>_out;
```

Note that this yields a solution with an unbounded counter variable. If one needs to avoid potential wrap-around, the alternate bounded counter implementation presented in Section 6 can be used.

Locks: Declare a private static Object `clusterCounterLock` to use for implementing exclusive access to counter variables.

```
private static Object
  clusterCounterLock = new Object();
```

Awaits: For each $\langle$**await** $B \rightarrow S\rangle$ at the entrance of region *rname* (**await** statements at the exit of a region are treated identically – only `exit` is used in the generated names for methods and variables), define one static public (non-synchronized) method named *rname*`$enter`, one static private method named `check$`*rname*`$enter`, and declare one static private variable of type Object named `condition$`*rname*`$enter` to implement specific notification.

We have the following declaration for the specific notification lock.

```
private static Object
  condition$<rname>$enter = new Object();
```

Public method *rname*`$enter` is defined as follows.

```
public static void <rname>$enter() {
  synchronized (condition$<rname>$enter) {
    while (!check$<rname>$enter())
       try {
          condition$<rname>$enter.wait();
        } catch (InterruptedException e){}
    }
  /* add notify calls here (see below) */
}
```

Private method `check$`*rname*`$enter` is defined as follows where `<B>` and `<S>` are the **await** guard and increment statements, respectively.

```
private static boolean
  check$<rname>$enter() {
    synchronized (clusterCounterLock) {
      if (<B>) {
        <S>; return true;
      } else return false;
    }
  }
```

Atomics: For each $\langle S \rangle$ at the entrance of region *rname* (atomic statements at the exit of a region are treated identically – only `exit` is used in the generated names for methods and variables), define one static public (non-synchronized) method named *rname*`$enter`, as follows.

```
public static void <rname>$enter() {
  synchronized (clusterCounterLock) { <S>; }
  /* add notify calls here (see below) */
}
```

Notifies: For each await/atomic statement generated at the entrance for region *rname*, add

```
synchronized (condition$<rname'>$enter) {
  condition$<rname'>$enter.notify();}
```

at the point of the comments concerning *notify* in the generated await/atomic statement if `<rname'>_in` appears in the `NOTIFY` list in the coarse-grain solution for *rname* `ENTER`, and add

```
synchronized (condition$<rname'>$enter) {
  condition$<rname'>$enter.notifyAll();}
```

if `<rname'>_in` appears in the corresponding `NOTIFYALL` list. The analogous steps are taken for atomic/awaits at region exits.

5. EXAMPLE

Using our basic invariant patterns and composition techniques, we have solved a wide variety of challenging problems found in standard textbooks [2, 3, 13], and all artifacts associated with eight representative problems can be found on the project web-site. In this section, we illustrate the use of the synchronization patterns to specify a solution to the *sleeping (daydreaming) barber problem* given in many OS textbooks (here, we use the description from [2]).

The shop has a barber, a barber chair, and a waiting room with N chairs. When a barber finishes cutting a customer's hair, the barber fetches another customer from the waiting room if there is a customer, or stands by the barber chair and daydreams if the waiting room is empty. A customer who needs a haircut enters the waiting room. If the waiting room is full, the customer comes back later. If the barber is busy but there is a waiting room chair available, the customer takes a seat. If the waiting room is empty and the barber is daydreaming, the customer sits in the barber chair and wakes up the barber.

In a solution, we define two types of threads; a barber thread and customer threads. Let integer variable *numCustomers* keep track of the number of customers in the waiting room. A scenario describing the repeating sequential behavior of the barber thread is as follows:

B1 {assertion: no customer is in the barber room} *wait until* a customer is in the waiting room
B2 {assertion: met a customer} start cutting the customer's hair
B3 finish the hair cut and inform the customer
B4 *wait until* the customer leaves the barber room

A scenario describing the repeating sequential behavior of the customer thread is as follows:

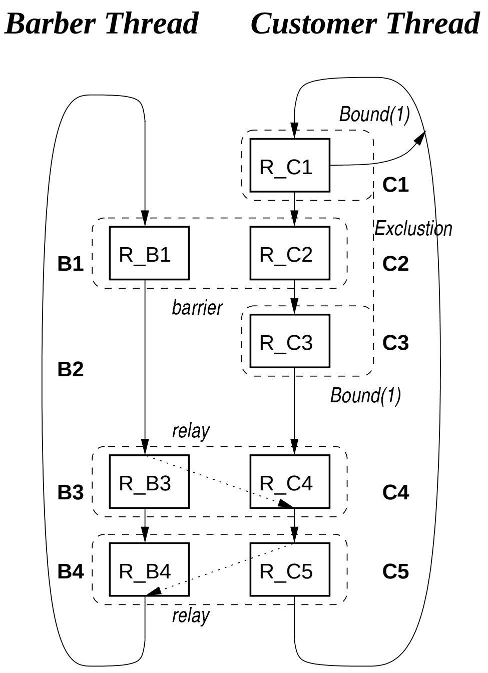


Figure 3: Sleeping Barber Synchronization Solution

C1 check if the waiting room is full (*numCustomers* $==N$) – if so, leave; else enter the waiting room (increment *numCustomers*).
C2 *wait until* the barber becomes free
C3 {assertion: met the barber} leave the waiting room (decrement *numCustomers*) and enter the barber room
C4 *wait until* the barber finishes the hair cut
C5 {assertion: hair cut is done} leave the barber room

In the barber's scenario, steps B1 and B4 constitute synchronization regions because they involve waiting on a customer (B1 waits on C2, B4 waits on C5). Step B3 is a synchronization region because it triggers the customer's exit. In the customer's scenario, steps C2 and C4 constitute synchronization regions because they involve waiting on the barber (C2 waits on B1, C4 waits on B3). Step C5 is a synchronization region because it releases the barber to look for another customer. Let these regions be denoted by R_{B1}, R_{B3}, R_{B4}, and R_{C2}, R_{C4}, R_{C5}.

We use a total of four clusters in the synchronization solution. The waiting and notification causalities listed above give rise to three clusters as described in Section 2: $\{R_{B1}, R_{C2}\}$, $\{R_{B3}, R_{C4}\}$, $\{R_{B4}, R_{C5}\}$. Cluster $\{R_{B1}, R_{C2}\}$ is synchronized by *Barrier*(R_{B1}, R_{C2}) since these two regions wait for each other. Clusters $\{R_{B3}, R_{C4}\}$ and $\{R_{B4}, R_{C5}\}$ are synchronized by *Relay*(R_{B3}, R_{C4}) and *Relay*(R_{C5}, R_{B4}), respectively since in each case the completion of the first region triggers the start of the second.

The fourth cluster $\{R_{C1}, R_{C3}\}$ is formed to guarantee mutually exclusive access to the counter *numCustomers* which is checked and incremented in C1 and decremented in C3. Thus, this cluster is synchronized by *Exclusion*(R_{C1}, R_{C3}) (which ensures that either $C1$ or $C3$ is vacant) and *Bound*$(R_{C1}, 1)$ and *Bound*$(R_{C3}, 1)$ (which ensure only one thread is accessing *numCustomers* in $C1$ and $C3$). Figure 3 displays the structure of the synchronization solution for the sleeping barber problem.

```
JAVA fragment:
   synchronized(this)
       this.x ++;

JIMPLE 3-address form for fragment:
   label1:
      nop;
      T$0 = this;
      entermonitor T$0;
   label2:
      T$1 = T$0.[Box.x:int];
      T$1 = T$1 + 1;
      T$0.[Box.x:int] = T$1;
   label3:
      exitmonitor T$0;
      goto label5;

BIR guarded assignments for fragment:
   loc s5:  when true
      do invisible  T_0 := this;
      goto s6;
   loc s6:  when lockAvailable(T_0.BIRLock)
      do  lock(T_0.BIRLock);
      goto s7;
   loc s7:  when true
      do  T_1 := T_0.x;
      goto s8;
   loc s8:  when true
      do invisible  T_1 := (T_1 + 1);
      goto s9;
   loc s9:  when true
      do  T_0.x := T_1;
      goto s10;
   loc s10:  when true
      do  unlock(T_0.BIRLock);
      goto s11;
```

Figure 4: Bandera's Intermediate Representations

6. SUPPORTING VERIFICATION

The approach described in this paper allows developers to be confident that the resulting system implementation satisfies correctness properties related to the specified synchronization policies provided that the synthesis process is correct. One approach to providing evidence of the correctness of the synthesis process would be to provide a proof of correctness of the synthesis algorithm. While potentially useful, this approach would not verify the implementation of the synthesis algorithm. We take an approach which analyzes the correctness of synthesized implementations directly by generating specifications of desired correctness properties that can be checked against the implementation. This kind of *checkable redundancy* provides an independent means of verifying that the synchronization implementation is correct.

Eliminating subtle errors in synchronization implementations is a significant advantage of our technique, however, other concurrency related errors may still be present in an application. We exploit synchronization specifications to construct *hybrid* models that blend synchronization behavior from the coarse-grain solution with the behavior of the functional core application code. This can result in dramatic reductions in the cost of verifying properties of SyncGen synthesized programs.

6.1 Software verification via model checking

We support verification of SyncGen synthesized programs by extending the Bandera toolset. Bandera [8] is a framework for translating Java source code into a finite-state model encoded in the input format of existing model checking tools, such as SPIN [17]. To be effective in combating the exponential complexity of model checking, Bandera provides support for reducing the number of states in the finite-state model while retaining the ability to reason about correctness properties. Bandera does this by providing different automated program analyses and transformations, such as program slicing [14] and data abstraction [10], that are aimed at state-space reduction.

Bandera is organized like an optimizing compiler that represents the program in a series of different formats that are amenable to different kinds of analyses. Bandera translates a Java program into a 3-address byte-code representation, called JIMPLE [26], to which traditional compiler optimizations can be applied, then it converts the JIMPLE representation to an asynchronous transition system model expressed using guarded-assignments, called BIR, and finally it converts the BIR representation to the model checker input format. Figure 4 gives a small fragment of Java code and its JIMPLE and BIR representations. The mappings between representations are, for the most part, straightforward. The only subtlety is the use of guards, i.e., the `when` clauses, in BIR to capture the semantics of JVM primitives, such as `entermonitor`. We note that BIRs guarded-assignment statements have the same semantics as the **await** statements introduced in Section 3.

6.2 Checking synchronization implementations

If one considers the coarse-grain solution as described in Section 3 it is immediately obvious that traditional model checking of finite-state systems cannot be used to verify synchronization implementations. The use of unbounded region entry/exit counters means that in principle those counters may yield an infinite state space.

One can use model checking as a thorough form of testing by applying it to a portion of the system's state space. Bandera allows users to perform *bounded model checking* for the portion of state space where values of designated variables stay within a specified subrange. Figure 6 gives the number of states explored during bounded model checks of the *Fine* grain solution for both the reader-writer (denoted RW in the figure) and Barber examples where *Unbounded* counters are used but the model check is restricted to states where the counters have values less than 4. In these cases, the checks were for deadlock and for synchronization policy related invariants encoded as assertions.

Such bounded model checks can provide evidence that the synchronization implementation is correct, but if one desires complete verification of correctness, the system's states must be explored exhaustively. One approach, which we mention below, is to attempt to abstract the unbounded counters to range over finite data domains. An alternative approach is to design the coarse-grain solutions for verification by using *bounded* counters that range over a, usually small, finite domain. We describe this approach for the bound and exclusion patterns below; bounded counter solutions for the rest of the patterns are given in [9].

Bound: The original global invariant for *Bound*(R, n) is $\mathit{In}_R - \mathit{Out}_R \leq n$. To adapt this to a bounded representation, we use a variable B to hold the value of $\mathit{In}_R - \mathit{Out}_R$. Thus, the adapted global invariant is $I_{bounded} = B \leq n$. In the original unbounded counter solution, the guard of the region entrance is $\mathit{In}_R + 1 \leq \mathit{Out}_R + n$. This is adapted to $B \leq n - 1$; the guard of the region exit is $true$. Unbounded counter operations In_R++ and Out_R++ are adapted to B++ and B- -, respectively.

We can verify the correctness of the bounded counter guards by calculating the weakest preconditions of bounded counter operations: for the region entrance

$$\begin{aligned} \text{wp}(B\text{++}, I_{bounded}) &= \text{wp}(B\text{++}, B \le n) \\ &= B + 1 \le n \\ &= B \le n - 1 \end{aligned}$$

for the region exit

$$\begin{aligned} \text{wp}(B\text{--}, I_{bounded}) &= \text{wp}(B\text{--}, B \le n) \\ &= B - 1 \le n \\ &= B \le n + 1 \\ &= true \quad \textit{...since } B \le n. \end{aligned}$$

Exclusion: The original invariant for *Exclusion*$(R_1, R_2, \cdots, R_n)$ is $\vee_{i\in\{1,\ldots,n\}}$*OnlyOneOccupied*$(i,n)$. where *OnlyOneOccupied* represents a formula that holds in a state in which threads are only in one region R_i (as defined in Section 2.2). To adapt the definition of *OnlyOneOccupied*, we use a family of counters $E_i (1 \le i \le n)$ where E_i holds the value of $In_i - Out_i$. Thus, the adapted definition of *OnlyOneOccupied*(i,n) is

$$\begin{array}{lll} & (E_1 = 0) & \wedge \quad \cdots \\ \wedge & (E_{i-1} = 0) & \wedge \quad (E_{i+1} = 0) \\ \wedge & \cdots & \wedge \quad (E_n = 0). \end{array}$$

Using the orginal version of *OnlyOneOccupied*, the unbounded counter region entrance guard for R_i is *OnlyOneOccupied*(i,n). Thus, the bounded counter version is obtained by simply using the adapted version of *OnlyOneOccupied*. The original region exit guard is *true*, and thus it remains unchanged in the bounded counter version.

We can verify the both of these guards by calculating the weakest preconditions of bounded counter operations following the same pattern as in the case for *Bound* above.

The chief advantage of the bounded counter solution is that exhaustive consideration of all possible execution states of a synthesized synchronization implementation can now be analyzed. In addition to this, comparison of state space sizes for the unbounded and bounded checks of the *Fine* grain solution of the 4 thread barber system in Figure 6 illustrates that the use of bounded counters can reduce the number of states. The reduction, however, appears rather modest, around 20%, and it is doubtful that it will mitigate the exponential growth of the state space with increasing numbers of components.

6.3 Hybrid model-extraction

A large body of recent work in model checking software, e.g., [4, 10, 27], has focused on identifying ways in which the low-level details of system descriptions and implementations can be abstracted. In principle these approaches attempt to recover a high-level model of the program that preserves behavioral properties that are relevant to a specification that is to be checked. Fully-automated versions, e.g., [4, 27], of such abstraction techniques generate very large numbers of sub-problems whose solutions are used to identify the high-level abstraction. Those sub-problems typically involve reasoning about arithmetic and consequently their solution requires the use of theorem-proving techniques. The number of sub-problems can grow extremely rapidly and this is a limiting factor in scaling these approaches. Semi-automated techniques, e.g., [10], require the user to help in the identification of abstractions, consequently these approaches may be even less scalable than the theorem-prover based techniques.

In our approach, we start with the high-level abstract model for the synchronization parts of the concurrent system and synthesize its implementation. Thus, confidence in the correctness of the synchronization code synthesis process can be leveraged to reduce the cost of model checking SyncGen synthesized applications by applying the high-level abstractions on hand.

```
Bounded Counter Coarse-grain Solution:
   CLUSTER: RW
   VARIABLES: E1_0, E1_1, B1;
   REGION: Reader
      ENTER: <AWAIT E1_1 == 0 --> E1_0++;>
      EXIT: <E1_0--;>
   REGION: Writer
      ENTER: <AWAIT E1_0 == 0 && B1 == 0
                    --> E1_1++; B1++;>
      EXIT: <E1_1--; B1--;>

JIMPLE Synchronization Calls:
   Label0:
      staticinvoke [RW.Reader$enter():void]();
      ...
      staticinvoke [RW.Reader$exit():void]();
      goto label1;

Hybrid BIR Model:
   loc s26:  when true
             do invisible  TEMP_2 := Reader__this;
             goto s27;
   loc s27:  when (E1_1 == 0)
             do  E1_0 := (E1_0 + 1);
             goto s28;
   loc s28:  when true
             ...
   loc s38:  when true
             do  E1_0 := (E1_0 - 1);
             goto s38;
```

Figure 5: Model Extraction Artifacts for Readers/Writers (excerpts)

Program	*Model*	*Counters*	*Threads*	*States*
RW	Fine	Unbounded	5	1599250
RW	Hybrid	Bounded	5	534
Barber	Fine	Unbounded	4 (2 Customers)	530514
Barber	Fine	Bounded	4 (2 Customers)	418527
Barber	Hybrid	Bounded	4 (2 Customers)	488
Barber	Hybrid	Bounded	5 (3 Customers)	4646
Barber	Hybrid	Bounded	6 (4 Customers)	46472
Barber	Hybrid	Bounded	7 (5 Customers)	500350

Figure 6: State-space Sizes for Model Checks

Our approach works in two phases integrated into Bandera's model extraction approach. First, parse the coarse-grain solution, e.g., as depicted at the top of Figure 5. Second, when traversing the JIMPLE representation of the program, e.g., as depicted in the middle of Figure 5, identify calls to synchronization region entry/exit implementations. In Bandera's source code model extraction process, the implementation of the fine-grain synchronization methods would be inlined and the resultant JIMPLE would be translated to BIR on-the-fly. For hybrid model extraction, instead of inlining those methods we splice BIR transitions into the model that encode the semantics of the region enter/exit commands in the coarse grain solution, e.g., as depicted on the bottom of Figure 5.

The data in Figure 6 illustrates the effectiveness of this approach in reducing the state space of systems for model checking. The reader-writer example, denoted RW, exhibits a state space reduction of a factor of 2994 and the Barber example exhibits a reduction of a factor of 1086 when using the *Hybrid* models. This preliminary data suggests that dramatic state space reductions are possible using this approach and that this reduction will vary with the application. The RW example exhibits a large reduction because the functional core code has very little data. The hybrid model eliminates the *lock* objects and the statements that manipulate those locks to implement the conditional waiting and notification in the fine grain solution since that behavior is implicit in the semantics of **await** statement guards. Consequently, the ratio of synchronization related data and control states to functional core code data and control states is very high for the RW example which leads to the large state space reduction. Relative to the RW example, the functional core code for the Barber example has a larger amount of data so the state space reduction is smaller.

Despite the significant reductions, it is clear that the state space of the hybrid model will still grow exponentially, as illustrated by the scaling of Barber example. Our hope is that slowing the rate of growth of the state space will allow scaling of model checking of hybrid models for systems well-beyond the point at which their implementations can be model checked, as was the case with the Barber example.

It is important to note, that the technique described in this paper does not address the potentially huge state space that may arise from the core functional code. We believe that ongoing work on abstraction techniques may provide a means of addressing that problem. On the other hand, our approach provides one way of abstracting synchronization code. Synchronization is not currently treated by automatic abstraction techniques, in part because of the lack of appropriate theories that would be needed to adapt theorem prover based techniques.

7. RELATED WORK

The integration of synchronization and object oriented programming has been studied extensively in the past decade. Several extensions to languages such as Java and C++ have been proposed with additional synchronization primitives to enhance flexibility and expressiveness. For example, several languages allow specification of conditions under which a method invocation can be accepted or blocked [19]. The notion of $coordinator$ in [19] allows a designer to specify the conditions under which a method may be invoked and the code to be executed on entry and exit of that method. The Composition filters [1] and Synchronization rings [16] models are similar in nature and define conditions to block or allow method invocations. [1] and [16] show how filters or rings can be composed in a layered fashion, allowing modular specification of synchronization aspects. implements inter-object synchronization via synchronizers. [19] also allows coordinators to control more than one object. The frameworks discussed above can be classified as following the bottom-up approach: they provide the language mechanisms to the designer to program synchronization. Our approach, on the other hand, is top-down. We start with high-level language independent specification of synchronization requirements and automatically derive the synchronization code. Hence, our work can be viewed as orthogonal and complimentary to the bottom-up approach.

[20] presents a framework for incorporating synchronization constraints into already developed code. These synchronization specifications are in terms of finite state machines and they employ a synchronous model that allows immediate distribution of the global state information. [6] presents a similar approach wherein *modal processes* are used to specify the legal combination of states of different processes. The system is implemented via a controller that controls the transitions between the states of the individual processes to ensure that illegal combinations are not reached.

The idea of using global invariants to specify synchronization policies is not new [2, 5]. Our approach differs from this work in several ways. [2, 5] present many coarse grained solutions in which functional codes are embedded within **await** statements. Our approach cleanly separates development of functional code and synchronization (aspect) code and weave them in a simple way. Solutions in [2] use many types of counters. Our approach successfully abstracts these counters by only two types of counters, In and Out. We have developed various synchronization patterns and their solution global invariants and demonstrated effectiveness of composition of such global invariants. We have also automated the translation from synchronization specifications to Java.

As mentioned in Section 6, model checking of software, and the abstraction of software to make such checking tractable, is an active area of research. Code can be viewed as a very detailed specification and this level detail leads to extremely large state spaces. For this reason, and to provide feedback in early phases of development, many researchers have advocated the application of automated verification techniques to requirements, e.g., [15], and software design, e.g., [11, 18], descriptions. This kind of support can be very useful, but without evidence of the conformance of designs with code they don't provide evidence of the correctness of the executable artifact. While work on automated refinement checking, e.g., [7], provides a kind of conformance checking, there is still the problem of relating the lowest level design to an implementation in a modern programming language like Java. In our approach, this design-code conformance is guaranteed by the code synthesis process so we are free to exploit the design information.

8. CONCLUSION

In this paper we describe a methodology for synthesizing correct implementations of synchronization policies from high-level specifications. Mizuno has shown that this methodology is broadly applicable to real synchronization problems when applied by hand [22, 23]. The main contributions reported on this paper are the development of the SyncGen tools that: automate the synthesis process, provide checkable redundancy for verifying the correctness of synthesized implementations, and exploit synchronization specifications for state-space reduction of general correctness properties.

SyncGen automates the synthesis of an intermediate representation of synchronization behavior, expressed as global invariants, through the use of automated decision procedures or, when users specify synchronization policies using patterns, through template instantiation. Backends for different languages and run-time environments, such as Java, C++/POSIX, and C/CAN based concurrent systems, have been developed that generate implementations from this intermediate representation and integrate the resulting synchronization code with the functional application code. These backends can also introduce redundant invariant specifications that can be checked using, for example, the Bandera tools. The Bandera tools themselves have been adapted to accept the functional core code and the intermediate-level synchronization specification then extract a model that represents their composite behavior.

Preliminary experience with the SyncGen tools on a variety of examples (see `www.cis.ksu.edu/saves`) is very encouraging. More experience is needed, however, to understand the breadth of applicability of this approach. Toward this end we are extending the methodology and SyncGen in several directions with the goal of evaluating its effectiveness by reengineering embedded vehicle control applications. This ongoing work includes: supporting the propogation of exceptions thrown in the core code past region boundaries, supporting group forming synchronization patterns using specific notification, adapting synchronization code generation to target available middle-ware platforms, such as TAO [25], and more tightly integrating our synchronization code generation with scheduling algorithms to support real-time applications.

Acknowledgements

The authors would like to thank Gurdip Singh and Mitch Neilsen for the many helpful discussions and comments related to this project. This work was supported primarily by DARPA/ITO's PCES program through AFRL Contract F33615-00-C-3044, with additional support from NSF under grants CCR-9703094, CCR-9708184, CCR-9896354 and CCR-9901605, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564, and from the Formal Verification of Integrated Modular Avionics Software cooperative agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

9. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In *Object-based distributed processing, Lecture Notes in Computer Science 791*, 1993.

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[3] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[4] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.

[5] A. Bernstein and P. Lewis. *Distributed Operating Systems and Algorithms*. Jones and Bartlett, 1993.

[6] P. Chou and G. Borriello. An analysis-based approach to composition of distributed embedded systems. In *Proc. of the International Workshop on Hardware/Software Codesign*, 1998.

[7] R. Cleaveland. The concurrency workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, Jan. 1993.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[9] X. Deng. Tool-support for invariant-based specification, synthesis, and verification of synchronization in concurrent Java programs. Technical report, 2001.

[10] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[11] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 152–163, Sept. 2001.

[12] D. Gries. *The Science of programming*. Springer-Verlag, New York, 1981.

[13] S. Hartley. *Concurrent Programming - The Java Programming Language*. Oxford University Press, 1998.

[14] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4), 2000.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[16] D. Holmes, J. Noble, and J. Potter. Aspects of synchronization. In *Proceedings of the Twenty-Fifth Conference on the Technology of Object-Oriented Languages and Systems (TOOLS Pacific '97)*, 1997.

[17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[18] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[19] C. Lopes and C. Kiczales. D: A language framework for distributed programming. In *Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center*, 1997.

[20] G. Matos, J. Purtilo, and E. White. Automated computation of decomposable synchronization conditions. In *IEEE High–Assurance Systems Engineering Workshop*, 1997.

[21] M. Mizuno. A structured approach for developing concurrent programs in Java. *Information Processing Letters*, 69(5):233–238, Mar. 1999.

[22] M. Mizuno. A pattern-based approach to developing concurrent programs in UML: Part 1. Technical Report 2001-2, Kansas State University, Department of Computing and Information Sciences, 2001.

[23] M. Mizuno. A pattern-based approach to developing concurrent programs in UML: Part 2. Technical Report 2001-3, Kansas State University, Department of Computing and Information Sciences, 2001.

[24] I. S. Organization. *11898 Road Vehicles – Interchange of digital Information – Controller area network (CAN) for high speed communication*, 1995.

[25] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), Apr. 1998.

[26] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[27] W. Visser, S. J. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the 3rd Workshop on Formal Methods in Software Practice (FMSP-00)*, pages 3–12, Aug. 2000.