

Separating Synchronization Concerns with Frameworks and Generative Programming

Scott D. Fleming, R. E. K. Stirewalt, Laura K. Dillon, and Beata Sarna-Starosta
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan USA 48824 {sdf, stire, ldillon, bss}@cse.msu.edu

Abstract

Thread synchronization in object-oriented systems is difficult to implement, in part, because traditional synchronization mechanisms are low level and, in part, because synchronization code is cross cutting and usually interleaved with functional code. In prior work, we addressed the first of these problems with a compositional model of mutual exclusion called *Szumo*, that is customized to the needs of strictly exclusive systems. This paper extends our prior work to address the second problem. We present SzumoC++, an approach to building strictly exclusive systems in C++ that affords a high degree of abstraction and separation of concerns. Development in SzumoC++ proceeds from an explicit design model, which is assumed to have been validated. Our approach uses object-oriented frameworks and a custom specification language to support the development of separate functional and synchronization modules which retain a high degree of design transparency. By virtue of this transparency, and with the aid of generative programming techniques, these separately implemented modules may be composed in a consistent manner. By virtue of being based on a Szumo design, we expect the resulting systems to be robust under change.

1 Introduction

Thread synchronization in object-oriented (OO) systems is a global and cross-cutting concern, which is difficult to implement and is resistant to clean separation from the primary functional logic of programs. While much progress has been made in the development of tools and frameworks for separating the synchronization from the functional logic of a program (e.g., [18]), synchronization concerns are difficult to modularize in such a manner that local changes to the functional logic engender commensurately local changes to the synchronization logic. Consequently, accommodating a local change or extension to the functional logic could require a redesign of the program's synchronization logic. This problem is especially vexing during software maintenance and is detrimental to the construction of open and adaptable systems. Our work aims to support the development and long-term maintenance of multi-threaded OO systems that are robust under change. We believe this goal can be achieved if the methods used to separate synchronization from functional logic are based on clean compositional design models, and if the separated synchronization and functional modules exhibit a high degree of transparency with respect to the design. This design transparency enables both the automated composition of and a form of consistency checking between the functional and synchronization modules.

Previously, we developed a compositional model of synchronization called the Synchronization Units Model (Szumo) [2, 25]. Szumo trades generality for compositionality by focusing on the category of *strictly exclusive systems*, in which multiple threads compete for exclusive access to dynamically changing sets of shared resources. This narrowing of focus was motivated by the observation that many applications fit well in this category and that, in such cases, we can exploit a clean, compositional model of design and verifi-

tion [22, 9]. Examples of strictly exclusive systems include extensible web servers and interactive applications with complex graphical user interfaces.

Our previous implementation of Szumo involved an extension of an existing programming language (Eifel) with new constructs that directly express the features of a Szumo design [1, 25]. Using this extended language, we were able to show how Szumo supports modular descriptions of the synchronization concern, raising the level of abstraction at which synchronization concerns are expressed, and automating implementation and enforcement of these concerns. A case study involving maintenance and evolution of a web server, based on the architectural design of Apache, demonstrated that a well-designed Szumo application could be safely modified and extended without requiring redesign of the original synchronization concern [3]. However, this implementation of Szumo lacked facilities for separating synchronization logic and functional logic to the extent achievable in approaches such as [4, 8, 18]. Upon reflection, we realized that an implementation of Szumo based solely on a language extension could not easily accommodate such facilities, especially in a modern production language such as C++.

This paper describes a new approach to programming Szumo applications that overcomes these deficiencies. The approach builds on explicit models of a Szumo design and a new programming system, called SzumoC++. A prototype implementation of our programming system can be found at:

<http://www.cse.msu.edu/sens/szumo/szumoc++.tar.gz>.

SzumoC++ uses a novel combination of plug-compatible OO frameworks and generative programming to impose upon the separated functional and synchronization modules a level of design transparency that enables the safe and automatic composition of concerns. The result simplifies programming of the synchronization concern and supports a separation of functional and synchronization concerns that inherits Szumo's support for robustness under change. In the remainder of the paper, we first give a background on the Szumo model and a sample Szumo design to use in a running example (Section 2). We then provide an overview of our approach (Section 3), deferring the details of its many components to later sections. We conclude with a comparison with related work (Section 8), and a discussion of lessons learned and possible directions for the future work (Section 9).

2 Szumo

Irrespective of how it is implemented (i.e., as a language extension or as part of a programming system), Szumo is a model of synchronization among entities in a strictly exclusive program. In this model, threads contend with one another for exclusive access to sets of *synchronization units*. Intuitively, the synchronization units in an application define indivisible units of sharing—at any time during execution, a thread is allowed to access either all of the objects contained in a synchronization unit or none of the objects contained in that unit [1, 25]. In lieu of code that invokes OS-level synchronization primitives, a synchronization unit declares one or more *synchronization constraints*, which specify the conditions under which the unit—acting as a *client*—needs exclusive access to units—acting as the client's *direct suppliers*—that the client holds references to. At run time, threads *negotiate* for exclusive access to suppliers in accordance with the constraints declared by any clients that they are executing. The declarative nature of synchronization constraints and this automated process of negotiation are key to Szumo's support for extension and maintenance [3]. As a background for subsequent sections, we now describe the details of a Szumo design, and provide an overview of the negotiation process.

2.1 Design Models

A key component of our approach is the use of an explicit model of a program's design, which is used both as a reference during implementation and also during the composition of separated concerns. Szumo design

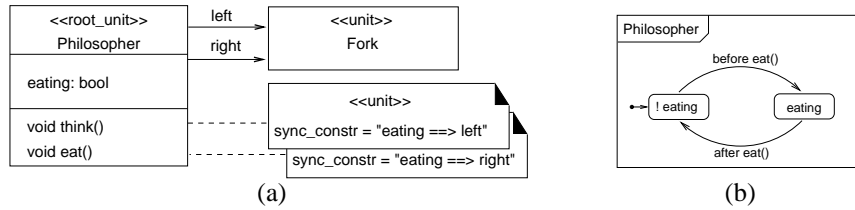


Figure 1: Diagrams depicting the unit-class and synchronization-state models of a Szumo design

models are expressed using a variant of the class and state diagram notations of UML 2.0 [21]. These models impose a modular structure upon and record key design decisions related to the synchronization concern.

An important design decision involves the granularity of sharing among the threads. Szumo allows application developers to choose the granularity of sharing by means of synchronization units, which are containers of (one or more) program objects. Synchronization units themselves resemble objects in that they may encapsulate state, provide operations, and reference other synchronization units whose operations they use to implement their own. When a program object is created, it is deployed to exactly one synchronization unit, where it remains throughout its lifetime. A thread that holds exclusive access to a synchronization unit, holds exclusive access to all objects contained within this unit.

Synchronization units are identified with instances of types called *unit classes*. In addition to standard operations, a unit class C may declare *unit variables*, *condition variables*, and synchronization constraints. Unit variables reference synchronization units that serve as direct suppliers to units of type C . Condition variables are boolean-valued variables in a unit class. The synchronization constraints specify the suppliers to which a client unit requires exclusive access based on the values of the unit’s condition variables and unit variables. We say that a client unit c *entails* a direct supplier s when c requires exclusive access to s , and refer to the set of all direct suppliers that c entails at any given time as c ’s *entailment*.

To illustrate these ideas, Fig. 1 depicts the models used to record the design of a Szumo solution to the familiar dining philosophers problem, which we will use as a running example in this paper. Fig. 1(a) depicts the *unit-class model*, which is just a UML class diagram with the following extensions. The stereotypes $\langle\langle\text{unit}\rangle\rangle$ and $\langle\langle\text{root_unit}\rangle\rangle$ designate unit classes whose instances are, respectively, units that are passive and can be shared among multiple threads, and units that serve as non-shared thread “roots”. We show condition variables as class attributes, unit variables as directed associations, and synchronization constraints as limited propositional formulas over condition variables and unit variables, formed using the *entailment operator* “ \Rightarrow ”. The set of synchronization constraints for a client class is given as a value for the “sync_constr” tag associated with that class. Thus, Fig. 1(a) shows that philosopher units execute in different threads (i.e., serve as root units) and perform operations on fork units bound to their unit variables, *left* and *right*. Fork units are passive and may be shared. Associated with each philosopher, condition variable *eating* signifies when the philosopher needs exclusive access to its forks. The philosopher’s synchronization constraint asserts that, if *eating* is true, the philosopher entails the fork units bound to *left* and *right*.

A *synchronization-state model*, which is depicted using a UML state diagram (Fig. 1(b)), shows how operations that a unit performs affect the values of its condition variables. Transitions are annotated with events that designate when they are taken. States are labeled with boolean expressions over the unit’s condition variables, signifying possible valuations of the variables in the states. An arrow with no source state marks the initial state. Thus, when a philosopher starts execution, its *eating* variable is false. Immediately before the philosopher invokes its *eat* operation, *eating* becomes true, and immediately upon return of this operation, *eating* becomes false. Together, the unit-class and synchronization-state models document that a philosopher entails its forks while executing an *eat* operation.

The diagrams in Fig. 1 show how we represent functional and synchronization concerns in a Szumo design. Although mixed, the representations of both concerns are highly abstracted in order to document how the concerns affect one another in a precise, but intuitive, fashion, and to enable generation of behavioral

models that can be analyzed for concurrency errors. A developer uses these behavioral models in verifying that a Szumo design exhibits necessary safety and liveness properties [22], and in generating visual traces of anomalous behaviors, such as deadlocks [9]. By such means, she can debug the design before implementing either of the concerns.

2.2 Negotiation

The set of units that a Szumo thread needs to access can be inferred at run time. For instance, from the diagrams in Fig. 1, we infer that a thread needs only its root unit, except when executing the root's `eat` operation, in which case it needs the `Fork` units referenced by the root's `left` and `right` unit variables. More generally, a thread needs all units in which it is executing, as well as any unit entailed by a unit that it needs.¹ The conjunction of the synchronization constraints associated with the units that a thread needs defines the thread's synchronization contract. The contract changes dynamically as the thread modifies condition variables and unit variables of the units in which it executes. Whenever a thread's contract changes, its contract must be (re)negotiated.

The semantics of contract negotiation is based on the concept of a *realm*, which is a set of synchronization units associated with a thread. Each thread has its own realm. When executing, a thread is allowed to access all units in its realm, and prevented from accessing any units outside its realm. To guarantee mutual exclusion, the realms of different threads must always be disjoint; thus, no unit is permitted to simultaneously be in the realm of more than one thread.

Changes in a thread's contract may result in the thread holding units that it no longer needs or needing units that it does not hold. When a thread's realm contains exactly the set of needed units, we say that the realm is *complete*; otherwise we say it is *damaged*. A thread with a damaged realm blocks until the realm is made complete. To make a thread's realm complete, unneeded units are first *migrated* out of the realm. Then, if no other thread holds any of the units needed by the thread, all units that the thread needs but does hold are atomically migrated into the realm; otherwise, the thread blocks until its realm can be completed (i.e., no needed units are held by other threads).

From a user's perspective, a thread executes within a unit in its (complete) realm until it performs an operation that modifies the values of the unit's condition variables or unit variables, thereby damaging the realm. Consider, for instance, a thread t whose initial realm contains only a philosopher unit p in its initial state, in which $p.eating$ is false. Then, the entailment of p is an empty set, and so t 's realm is complete. However, before invoking the `eat` operation, $p.eating$ becomes true, thereby affecting p 's entailment and damaging t 's realm. To complete t 's realm, the newly entailed fork units referenced by $p.left$ and $p.right$ must be migrated into the realm. If neither of the fork units is held by another thread, they are atomically migrated into t 's realm, and t continues its execution. Conversely, if either unit is in some other realm, t blocks, with its realm damaged, until both units become available; at which time they are atomically migrated into t 's realm, and t continues its execution.

3 Overview of SzumoC++

Having analyzed and verified a Szumo design, a developer would naturally like the implementation to be as faithful to the design as possible. However, to enable analysis and verification, a Szumo design must indicate how functional and synchronization concerns affect one another. Mixing these concerns in an implementation obstructs understanding of the code and complicates testing and maintenance. Thus, when concerns are mixed in the design model, the goals of separating functional and synchronization concerns and of producing

¹Formally, we define the needs of a thread as the smallest set of units that contains the set of units a thread is executing and that is closed under the *entails* relation. This inductive definition of needs based on the entailment closure is necessary for local synchronization constraints to affect the global synchronization behavior.

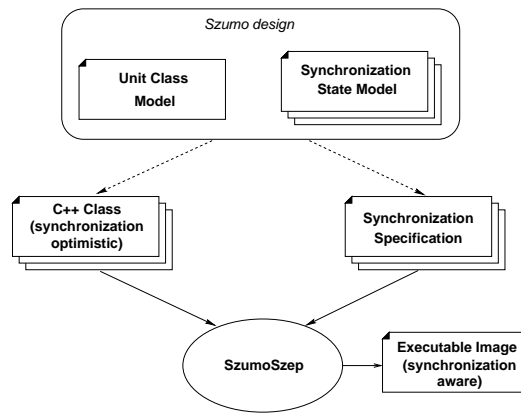


Figure 2: Overview of SzumoC++ approach

a transparent implementation of the design appear to be at odds with one another. SzumoC++ addresses this problem, effectively supporting both goals.

Assuming that the developer is confident that the synchronization concern will be correct if implemented and integrated with the functional concern in accordance with a design, separating the programming of these concerns is justified. We support this separation of implementations while providing a high level of design consistency and transparency using a two-phase approach, which is illustrated in Fig. 2. Starting from a Szumo design, the developer first programs and debugs the functional logic, without worrying about synchronization. She then separately programs the synchronization logic and weaves this logic with the functional logic using an automated tool. In the figure, programming tasks are depicted using dashed arrows; whereas data flows into and out of the automated tool are depicted using solid arrows. To enforce design transparency and to guard against inconsistencies that may arise by virtue of separating concerns, SzumoC++ includes a pair of white-box OO frameworks [15], called `SimpleFrame` and `SzumoFrame`, and a generative programming tool, called `SzumoSzep`.

In the first phase of our two-phase approach, the developer uses `SimpleFrame` to code up the functional logic of the program so that it reflects the structure of the unit-class model. She specializes framework classes to produce C++ unit classes, whose instances will become the synchronization units in the completed program. In this phase, the developer simply assumes that clients will have exclusive access to their direct suppliers whenever necessary and writes code to implement only the functional logic. We refer to the program developed in this phase as being *synchronization optimistic* because it is written under this optimistic assumption.

`SimpleFrame` classes are essentially placeholders that stand in for their `SzumoFrame` counterparts. From the programmer’s standpoint, classes in `SimpleFrame` are plug compatible with classes in `SzumoFrame` in the following sense: An application class that inherits from a `SimpleFrame` class can be reparented to inherit from a `SzumoFrame` class of the same name without having to modify any of the programmer’s code. Thus, a program that instantiates `SimpleFrame` can be ported to one that instantiates `SzumoFrame` without having to *undo* any of the programmer’s design decisions. Of course, to exploit the synchronization facilities provided by the `SzumoFrame` classes, it will be necessary to add code to the synchronization-optimistic program. This separation allows the programmer to perform early unit testing to check for errors in the functional logic. Locating sources of errors detected in this phase is simplified by not having synchronization code woven in. Moreover, the transparency of the synchronization-optimistic program with respect to the design could be verified, using either auditing procedures or through the facilities of a UML-based IDE. This transparency is essential to ensure that this program will compose correctly with the separately specified synchronization logic.

In the second phase of our approach, the developer writes *synchronization specifications*, which collec-

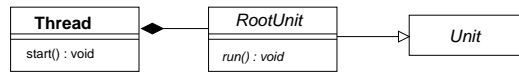


Figure 3: Class diagram for SimpleFrame

tively define the synchronization concern, and uses SzumoSzep to integrate this concern into the synchronization-optimistic program. A synchronization specification is supplied for each unit class in the design. It is written in a special notation that expresses the synchronization relevant aspects of a Szumo design in a highly transparent manner. SzumoSzep promotes synchronization-optimistic unit classes into classes that are *synchronization aware* by instantiating SzumoFrame rather than SimpleFrame and weaving in logic generated from the synchronization specifications to invoke operations defined in SzumoFrame. In addition to its generative and weaving capabilities, SzumoSzep provides a degree of consistency checking between the synchronization-optimistic program and the synchronization specifications. This capability derives from the design transparency afforded by our approach.

A key difference between our approach and aspect-oriented approaches to separating synchronization concerns, such as the D framework [18], is our reliance on an explicit design and the measures we take to ensure design transparency among the implementations of the separated concerns. Our approach is thus able to provide prescriptive guidance on how to design the functional logic to enable a clean separation of concerns and also to check for inconsistencies during composition. To our knowledge, no other approach can achieve this degree of separation of global synchronization concerns, even when restricted to only the category of strictly exclusive systems. Our use of a surrogate framework that allows a developer to “condition” the functional logic so that it may be plugged into a complex and feature-rich framework is also novel. The developer works directly with SimpleFrame in the first phase and SzumoSzep in the second. She never directly works with SzumoFrame.² Rather, SzumoSzep automatically instantiates SzumoFrame while weaving in the synchronization logic. We are not aware of other approaches that use plug-compatible OO frameworks and generative programming in this synergistic manner. The remainder of this paper describes the various components of our approach in more detail and reflects on the design of these components.

4 Synchronization-Optimistic Programs

We now elaborate this idea of a synchronization optimistic program, whose classes implement the “functional part” of a Szumo design. Recall that such a program assumes exclusive access to shared resources when they are needed without taking steps to guarantee this assumption. Synchronization optimistic programs are constructed using the facilities of an object-oriented framework called SimpleFrame, whose instantiation is guided by a Szumo design (Section 4.1). The resulting program incorporates the structure of this design but lacks any synchronization logic. Its classes are thus more amenable to understanding and unit testing; however, if executed, the entire program is likely to exhibit concurrency errors, such as data races. Such a program must then be *promoted* into one that is synchronization aware. Generally speaking, it is difficult to promote an arbitrary program with no synchronization logic into one that is synchronization aware (Section 4.2). However because a synchronization optimistic program incorporates the structure of a Szumo design, promotion can be completely automated, as explained in the sequel.

4.1 SimpleFrame by Example

SimpleFrame is an object-oriented framework that enables a programmer to easily construct synchronization-optimistic concurrent programs such that the objects are partitioned into synchronization units. Recall that

²Section 6 provides details about SzumoFrame and how it is instantiated for the reader who is interested, not just in how an application developer uses SzumoC++, but also in how it works. Readers who are not interested in these details may safely skip Section 6.

```

1 class Fork : public virtual Unit { ... };
2
3 class Philosopher : public virtual RootUnit {
4 public:
5     Philosopher(Fork* l, Fork* r, int i);
6     void eat();
7     void think();
8     virtual void run();
9     ...
10 private:
11     Fork* left; Fork* right; int id;
12 };
13
14 void Philosopher::eat()
15 { cout << left <<" and "<< right <<" are in use\n";
16   cout << this <<" is eating\n";
17   cout << left <<" and "<< right <<" are free\n";
18 }
19
20 void Philosopher::think()
21 { cout << this <<" is thinking\n"; }
22
23 void Philosopher::run()
24 { while (true) { think(); eat(); } }
25 ...

```

Figure 4: Synchronization-optimistic program

an object-oriented framework is an application skeleton that is fleshed out into a concrete application by a process called *framework instantiation*, which involves:

- designing classes that extend (i.e., inherit from) one or more framework classes,
- writing code that allocates and configures instances of these new classes and perhaps also instances of unextended framework classes, and
- writing or reusing a “main” program that cedes control to a driver, which is provided by the framework.

SimpleFrame provides three framework classes, Thread, Unit and RootUnit (Fig. 3). The Thread and RootUnit classes are similar to the Java API classes Thread and Runnable, respectively. SimpleFrame is instantiated as follows. First, the designer creates a collection of application classes that inherit and appropriately extend the framework classes Unit and RootUnit. This is a trivial exercise because a Szumo design clearly identifies which application classes are to be synchronization classes, and thus should inherit from Unit, and which should be root-unit classes, and thus should inherit from RootUnit. Next, she allocates instances of these application classes and of the framework class Thread and configures these instances appropriately. Finally, she writes a “main” program that calls into a framework function `init_runtime`, which causes the main thread to block until all activated threads have terminated.

Fig. 4 illustrates how to instantiate SimpleFrame for the dining philosophers problem. To designate that the forks are passive shared (collections of) objects, class Fork inherits from Unit. To produce philosopher units that may serve as the root of a thread’s realm, class Philosopher inherits from RootUnit and provides an implementation of `run` for a thread to invoke. It declares instance variables `left` and `right`, which will be bound to fork units to use when eating. The `run` method repeatedly invokes two local methods, first `think` and then `eat` (lines 23–24). Invoking `eat` on a philosopher appends a trace to the output stream (lines 15–17). In this case, the trace models the activity a philosopher performs to eat.

The simplicity of the methods in Fig. 4 owes to the fact that they express only synchronization-optimistic logic, without regard to the global context in which the object is deployed: A philosopher repeatedly thinks and then eats, emitting a trace modeling its activities; moreover, it performs these activities regardless of how many philosophers are created, how the philosophers are configured to reference forks, whether philosophers eat concurrently or sequentially, and so on. Of course, without proper synchronization, there is no way to

ensure that if two philosophers share a fork, they never eat at the same time. For example, the synchronization-optimistic program might generate a trace of the form

```
...
Fork 0 and Fork 1 are in use
Philosopher 0 is eating
Fork 1 and Fork 2 are in use
Philosopher 1 is eating
...
```

in which two philosophers are modeled as both eating with `Fork 1` at the same time. Synchronization logic must be introduced to prevent multiple threads from interleaving in this manner.

4.2 Obstacles to Promotion

Readers who are familiar with textbook solutions to the dining philosophers problem should appreciate the transparency of the code in Fig. 4. The code becomes much harder to read and understand when it includes synchronization logic. For example, in a common monitor-based solution, each fork is implemented as a monitor, which encapsulates a status variable that records availability and which supplies two methods, `up` and `down`, for philosophers to invoke before and after eating. Briefly, the `up` method either makes the fork unavailable or, if it is already unavailable, inserts the caller on the fork's *wait queue* and blocks; whereas the `down` method makes the fork available and, if the fork's wait queue is nonempty, also unblocks a waiting philosopher. Then, to prevent races on the forks, a philosopher calls `left.up` and `right.up` before the call to `eat` and `left.down` and `right.down` after the call. As long as the monitors are fair, this design is also fair. However, it does not necessarily prevent deadlock. A common strategy for preventing deadlock is to impose a total order on the forks and arrange that each philosopher calls `up` on the fork that comes first in this order and then calls `up` on the other fork.

Even in this simple example, the synchronization logic becomes complex. Much of this complexity stems from the low level at which synchronization is operationalized. The requirement that a philosopher needs exclusive access to both forks while eating is simple enough to state and understand. But implementing it requires defining a low-level protocol for how objects exchange status information in method calls, manipulate queues, and block and unblock.

Additionally, the code implementing this protocol is not confined to one module, but is spread throughout the code base. As a result, it is difficult to reason about. For example, to reason that the synchronization logic in class `Fork` is correct requires knowing properties of class `Philosopher` that are not typically documented in the class interface (e.g., that philosophers call a fork's `up` method before eating and its `down` method after eating and never call `up` twice without calling `down` in between). Thus, two key problems that contribute to the complexity of thread synchronization in object-oriented systems are: First, the synchronization logic is operationalized at a low level of abstraction. Second, the code that operationalizes it is cross cutting and interleaved with the functional code. SzumoC++ addresses both of these problems.

5 Synchronization Specifications

We now elaborate the notion of a synchronization specification, which defines that aspect of the synchronization concern that pertains to a particular unit class in a Szumo design. The specification declares a unit's entailment at a suitably high level of abstraction. In the sequel, we illustrate how the SzumoSzep tool uses synchronization specifications to promote syncopt unit classes into synchronization-aware unit classes. The resulting synchronization-aware program incorporates logic for ensuring that a thread does not access units outside its realm and a negotiation protocol that blocks a thread from executing until when all of its needs are met.

```

1 sync_spec Fork {};
2
3 sync_spec Philosopher {
4   unit left; unit right;
5   sync_pointcut phil_eats : call(eat);
6   condition eating {
7     init(false);
8     trigger before: phil_eats;
9     cancel after: phil_eats;
10  }
11  constraint {
12    eating ==> left;
13    eating ==> right;
14  }
15 };

```

Figure 5: Example synchronization specifications

Fig. 5 depicts synchronization specifications for the unit classes `Philosopher` and `Fork`. Each specification consists of a header, introduced by the keyword `sync_spec`, followed by a body, delimited by set braces. The header names the unit class to be promoted, and the body declares how operations affect the values of condition variables. The body of the synchronization specification for `Fork` is empty (line 1) because fork units have no condition variables or synchronization constraints. In contrast, the body of the synchronization specification for `Philosopher` formalizes the requirement that when a philosopher is eating, she needs exclusive access to both of her forks. In order to do so, it declares

- two unit variables, `left` and `right` (line 4);
- a *pointcut designator*, `phil_eats` (line 5);
- a condition variable, `eating` (line 6), an *initial-value clause* for `eating` (line 7), and two *transition clauses* specifying when the value of `eating` may change (lines 8 and 9); and
- two synchronization constraints (lines 12 and 13).

The notion of pointcut designators derives from the AspectJ language [16]. The other features are imported almost verbatim from Szumo design models.

The `unit` declarations name unit variables, which are assumed to have been declared in the unit-class model. The current version of SzumoSzep handles synchronization constraints of the form

$$\text{condVar} ==> \text{unitVar} \quad (1)$$

where `condVar` and `unitVar` stand for a condition variable and a unit variable, respectively. As with `unit` declarations, these constraints are imported directly from a unit-class model in a Szumo design.

Condition variable blocks are used to declare condition variables, which appear as class attributes in the unit-class model. The value of a condition variable is defined by an initial-value clause and zero or more transition clauses. When a synchronization unit is created, its condition variables are initialized with the indicated initial values. Subsequently, the value of a condition variable can change only before or after executing an operation selected by a pointcut expression appearing in one of the variable's transition clauses. SzumoSzep currently supports only pointcut designators that select method invocations, indicated by the pointcut expression `call(fun)`, where `call` is a keyword and `fun` is the method name. In a transition clause, `before` indicates that the value changes immediately before executing the operation and `after` indicates that the value changes immediately after; whereas `trigger` indicates that the variable becomes true and `cancel` indicates that it becomes false. A pointcut designator abbreviates a pointcut expression. For instance, the synchronization specification for `Philosopher` declares `phil_eats` (line 5), which is used in specifying the pointcuts in the transition clauses for `eating`. The first transition clause (line 8) declares that `p.eating` becomes true immediately before `p` calls `eat` (Fig. 4, line 14) and the second (line 9) declares that `p.eating` becomes false immediately upon return from the call, for a philosopher `p`.

As they are defined, the condition variables and unit variables in a unit represent the synchronization-relevant portion of a unit's state. They determine a unit's entailment, from which a thread infers the suppliers that it must hold exclusive access to in order to safely execute in the unit. When this portion of a unit's state changes, the thread executing the unit enters into a negotiation with other threads competing for exclusive access to shared suppliers.

6 SzumoFrame

Synchronization-aware programs perform two functions over and above those performed by their synchronization-optimistic counterparts: They check accesses to a synchronization unit to ensure that the unit is in the realm of the accessing thread, and they implement a protocol according to which threads negotiate for exclusive use of dynamically determined sets of synchronization units. `SzumoFrame` is an OO framework that encapsulates much of the logic for realm boundary checking and negotiation into framework classes, which are reused and extended in the construction of a synchronization-aware program. `SzumoFrame` is plug compatible with `SimpleFrame`, but it is also significantly larger in terms of the number of framework classes and methods and the number of hot spots that a programmer must fill in. To appreciate the value added by `SzumoSzep`, we now summarize in some detail what would be involved if developers had to manually instantiate `SzumoFrame`. Readers who are interested only in how to use `SzumoSzep` may safely skip over this section.

6.1 Behavior of Synchronization-Aware Program

In a synchronization-aware Szumo program, each thread is prevented from accessing any unit that is outside of its realm, and threads negotiate with one another for exclusive access to dynamically determined sets of synchronization units. The responsibility for realm-boundary checking is relegated to the synchronization units as follows: Each client unit must perform a realm-boundary check prior to invoking a method on a supplier. `SzumoFrame` provides a collection of class and function templates to automate these checks. For brevity, the examples we present in this paper do not illustrate their use; however, their application is straightforward and is handled automatically by `SzumoSzep`. Thread negotiation is implemented by a highly dynamic and decentralized collaboration, within which both thread objects and synchronization units play a role. To instantiate `SzumoFrame` is to design (or adapt) application classes so that their instances are capable of playing an appropriate role in this collaboration.

Each instance of the negotiation collaboration comprises one or more objects that play the `ThreadContext` role and one or more objects that play the `Unit` role. Thread context objects encapsulate the state and operations required to manage and evolve the realm of a thread, and unit objects encapsulate the state and operations required for a synchronization unit to be negotiated for. Thus, a synchronization unit may now be understood to be an object that is capable of playing the `Unit` role. Such an object must:

1. maintain a representation of its *synchronization-relevant state*,
2. be receptive to *entails* messages, returning a collection of references to other synchronization units, which this unit entails based on its current synchronization state, and
3. notify the thread that *holds* this unit whenever its synchronization state changes.

Unit and thread-context objects collaborate as follows: Notification (item 3) occurs when a unit object sends a *damage_realm* message to the thread-context object associated with the currently executing thread. This message indicates that the realm of the current thread might now be damaged and may thus be in need of repair. The thread-context object responds by attempting to repair the realm, which involves sending *entails* requests to unit objects to decide whether a realm is complete and to figure out which, if any, units may be released from the realm or which, if any, must be acquired.

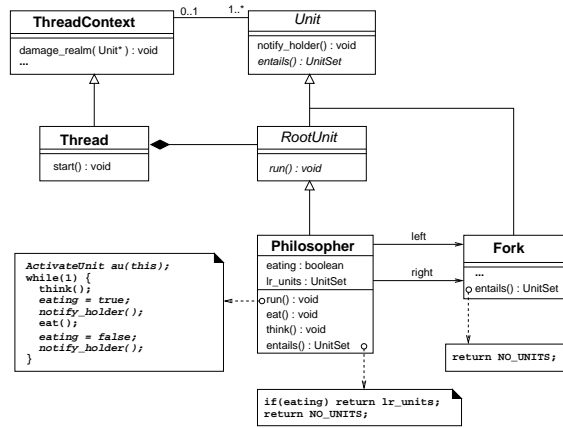


Figure 6: Dining philosophers instantiation of SzumoFrame

To assist in the development of objects that play these roles, SzumoFrame provides several framework classes and functions. The framework class `Unit` provides operations and hot spots for implementing objects that are capable of playing the `Unit` role. Class `Unit` declares an abstract operation `entails`, thereby making instances of any class that derives from `Unit` receptive to `entails` messages (item 2). It also provides a protected, non-polymorphic operation called `notify_holder`, which locates the thread context associated with the currently executing thread and sends it a `damage_realm` message (item 3). A synchronization class then extends class `Unit` by declaring instance variables to represent the synchronization-relevant state, maintaining this state and invoking `notify_holder` when it changes, and providing a method for the `entails` operation. For completeness, notice that the framework class `RootUnit` extends class `Unit`, adding the declaration of an abstract operation called `run`, and the framework class `Thread` extends class `ThreadContext`.

Before proceeding with an example, we should point out two characteristics of the design of SzumoFrame. First, all of the “traditional” synchronization mechanisms—e.g., mutex locks, wait and notify statements, are encapsulated in the two classes `Unit` and `ThreadContext`, hidden away from clients who wish to instantiate the framework. Second, the public interfaces of classes `RootUnit` and `Thread` exactly mimic the public interfaces of their `SimpleFrame` counterparts *assuming one ignores the public operations that are inherited from the base classes*. Together, these design decisions are key to separating concerns from Szumo users. To collaborate, objects that play the `ThreadContext` and `Unit` roles must be able to “see” operations that we wish to hide users who wish to instantiate the framework. By forcing developers to instantiate `SimpleFrame` and then use `SzumoSzep` to promote their code into an instantiation of `SzumoFrame` rather than designing against `SzumoFrame` directly, we enforce this separation of concerns.

6.2 Example Instantiation

Figs. 6 and 7 depict an instantiation of SzumoFrame for the dining philosophers example. All of the synchronization-optimistic code (Fig. 4) is preserved in the synchronization-aware version (Fig. 7). The differences are purely additive, and are called out explicitly with enclosing boxes in Fig. 7. We now briefly describe and explain these additions.

Recall that class `Unit` declares an `entails` operation but provides no method for it, as the method will vary depending upon the synchronization state and exclusion constraints of a given unit. Lines 2 and 3 implement an `entails` method for class `Fork`. In this case, the method is trivial because `Fork` objects provide no synchronization state. Thus, a `Fork` object responds to `entails` messages by returning the empty entailment `NO_UNITS`.

```

1 class Fork : public virtual Unit {
2     public: virtual const UnitSet& entails() const
3         { return NO_UNITS; }
4 };
5
6 class Philosopher : public virtual RootUnit {
7     public:
8         Philosopher(Fork *l, Fork *r, int i);
9         void eat();
10        void think();
11        virtual void run();
12        ...
13        public: virtual const Needs& entails() const
14            { if (eating) return lr_units;
15              return NO_UNITS; }
16        private:
17            Fork *left; Fork *right; int id;
18            private: mutable bool eating;
19            private: mutable UnitSet lr_units;
20            ...
21 };
22
23 Philosopher::Philosopher(Fork *l, Fork *r, int i)
24 : RootUnit(), left(l), right(r), id(i), eating(false)
25 { ActivateUnitConstructor auc(this);
26   lr_units.add(left); lr_units.add(right); }
27
28 void Philosopher::eat()
29 { ActivateUnit au(this);
30   cout << left <<" and "<< right <<" are in use\n";
31   cout << this <<" is eating\n";
32   cout << left <<" and "<< right <<" are free\n";
33 }
34
35 void Philosopher::think()
36 { ActivateUnit au(this);
37   cout << this <<" is thinking\n";
38 }
39
40 void Philosopher::run()
41 { ActivateUnit au(this);
42   while (true) {
43     think();
44     (eating = true, notify_holder(), eat(),
45      eating = false, notify_holder() );
46   }
47 ...

```

Figure 7: Dining philosophers synchronization-aware program

Class `Philosopher` declares a Boolean variable `eating` (line 18) that constitutes part of the synchronization-relevant state of a philosopher unit, and line 24 initializes this variable to false. The `entails` method for class `Philosopher` (lines 13–15) returns `lr_units`, a collection comprising references to the units bound to the `left` and `right` instance variables, when `eating` is true and the empty entailment when `eating` is false. By convention, each unit declares a variable for every possible set of unit references it may entail (except the empty set, which is provided by the `SzumoFrame` variable `NO_UNITS`). A `Philosopher` unit will always entail either nothing or the set containing both its `left` and `right` unit references. Class `Philosopher` declares (line 19) and initializes (line 26) a unit set called `lr_units` to represent the latter case. We use these pre-computed unit sets to ensure calls to `entails` are as efficient as possible.

Lines 25, 29, 36, and 41 each declare instances of an object (called `au`) whose lifetime coincides with the activation of the method in which they are declared. These objects add an *implicit synchronization constraint* on the object that is hosting the activation. Without implicit constraints, a situation might arise during a series of recursive calls between two units where one unit’s explicitly specified synchronization constraints dictate that access to the other unit is no longer needed when in fact it is. Every unit-class method (except `entails`) should begin by declaring such a variable.

Lines 44 and 45 update the synchronization-relevant state and notify the currently executing thread of state changes. This state is being maintained in two places in these lines. One is before the call to `eat`, where the condition variable `eating` is set to true, and the other follows the call to `eat`, where `eating` is set to false. Notice how these assignments to `eating` create the effect that while the method `eat` is executing, the `Philosopher`’s synchronization-relevant state reflects the fact that it is eating. Since assigning new values to one or more condition variables represents a change in synchronization-relevant state, such a sequence of assignments must be followed with a notification to the holder thread, which is accomplished in the code by calling `notify_holder`.

6.3 Connecting to Synchronization Specifications

By design, the signature of each framework class in `SimpleFrame` is a subset of the signature of its counterpart in `SzumoFrame`. This allows the insertion of a significant amount of synchronization logic into a synchronization-optimistic program by merely recompiling the program against `SzumoFrame` rather than `SimpleFrame`. However, to fully exploit the benefits of this separation, we need a fully-automated way to enhance a `SimpleFrame` instantiation with the kind of code that appears in the boxes in Fig. 7. This is where synchronization specifications come into play.

Each synchronization specification describes how to adapt a synchronization-optimistic unit class, hereafter the *adaptee*, into one whose instances are capable of playing the `Unit` role in `SzumoFrame` as follows. Condition variables in the synchronization specification engender boolean instance variables in the adaptee. Likewise, the synchronization constraints collectively engender an `entails` method, which consults these instance variables to determine entailments. Finally, the transition clauses engender logic for updating these boolean instance variables and for invoking the framework method `notify_holder` upon change. This logic (and also logic to invoke `notify_holder` whenever a unit variable is updated) must then be woven into the body of the existing methods of the adaptee. `SzumoSzep` performs both the code generation described here and also the weaving necessary for promotion.

7 SzumoSzep

As mentioned previously, `SzumoSzep` takes a synchronization-optimistic program and a collection of synchronization specifications and promotes the former into a synchronization-aware program (Fig. 2). `SzumoSzep` combines code generation and source-to-source translation capabilities with a standard C++ compiler to:

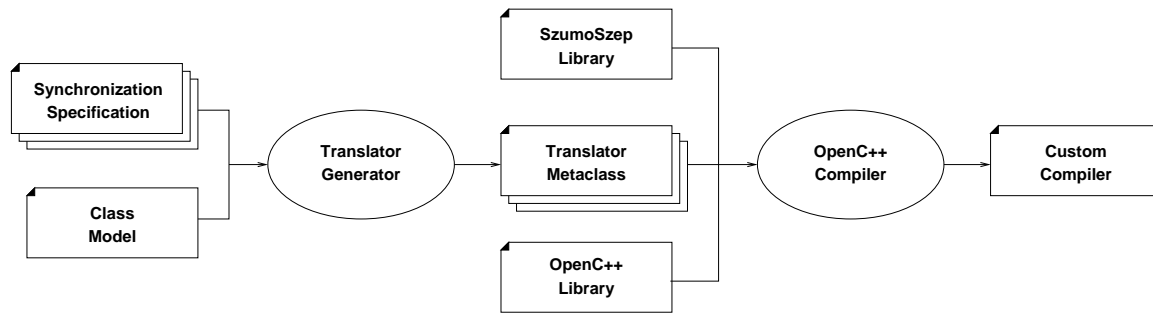


Figure 8: Generation of the custom compiler

1. type check synchronization specifications against a synchronization-optimistic program,
2. generate synchronization logic from the synchronization specifications and then rewrite the subject program into to incorporate this logic at the appropriate locations, and
3. compile and link the resulting program into an executable image.

The synchronization-optimistic program is assumed to be an instantiation of `SimpleFrame`. By contrast, the resulting program is compiled against the `SzumoFrame` classes, as opposed to the `SimpleFrame` classes, effectively promoting an instantiation of the latter framework into an instantiation of the former. We now briefly describe the `OpenC++` [6] tool, which we used to implement the translation (Section 7.1) before describing the architecture of `SzumoSzep` in detail (Section 7.2).

7.1 OpenC++

Numerous tools have been developed to assist programmers in inserting new code into an existing C++ program [6, 14, 24]. `OpenC++` is a tool for customizing a C++ compiler, including modest extensions to the language syntax. The custom compiler is a standard C++ compiler with a source-to-source translation phase interposed between preprocessing and compilation. One develops custom compilers by customizing the translator, which implements this intermediate translation phase.

The translator itself is designed for extension using ideas from object-oriented frameworks. During translation, each input feature (e.g., class, method, assignment statement) is parsed into an internal representation which is then forwarded to an object called a *metaobject*, whose operations are invoked to translate features of the particular input type. For example, having recognized and constructed an internal representation of a C++ class definition in the input, the translator transforms this representation by invoking operations on a *class metaobject*. In a similar manner, the internal representation of a method in the input is translated by invoking operations on a *method metaobject*. The framework classes in this extensible design are those from which metaobjects of the various kinds are instantiated; in `OpenC++` parlance, such classes are called *metaclasses*. `OpenC++` translators are thus customized by developing new metaclasses, which inherit from the framework metaclasses, and overriding one or more of the inherited operations with new methods.

7.2 SzumoSzep Architecture

`SzumoSzep` uses `OpenC++` to generate a custom compiler that, when invoked on the synchronization-optimistic program, will translate it to promote its `SimpleFrame` instantiation to a `SzumoFrame` instantiation before compiling against and linking with the `SzumoFrame` library. Fig. 8 depicts the process of generating this custom compiler. The *Translator Generator* reads in the synchronization specifications and a model

of the (synchronization-optimistic) classes of the subject program, and from these generates a collection of OpenC++ metaclasses, one for each unit class in the Szumo design. Each generated metaclass is customized to adapt a specific synchronization-optimistic class, using the adaptation strategy described in Section 6.3. OpenC++ then compiles these metaclasses (and various supporting library classes) to produce the custom compiler.

Notice that the Translator Generator requires an “as-built” model of the synchronization-optimistic classes. By “as-built,” we mean that the model reflects the implementation structure of these classes, including the names and types of all declared instance variables and any inheritance relationships. This model must be derived from the program itself; in fact, we harnessed OpenC++ to generate a program that extracts it automatically (not depicted in the figures). Moreover, the as-built model must be a consistent refinement of the unit-class model that was supplied in the Szumo design. Briefly, each a class in the as-built model must have a counterpart of the same name in the design model, and all inheritance relationships must be preserved. Classes in the as-built and the design model may have non-overlapping sets of attributes, as the attributes in the design model are used to define condition variables, which are a part of the synchronization concern and which should not appear in the synchronization-optimistic program.

Finally, the generated compiler is used to rewrite, compile, and link the synchronization-optimistic program, yielding a synchronization-aware program that instantiates `SzumoFrame`. Recall our goal was to completely separate synchronization concerns at the implementation level. The ability to compile a `SimpleFrame` instantiation against `SzumoFrame` contributes significantly to the accomplishment of this goal but cannot completely separate synchronization concerns because the classes that instantiate `SzumoFrame` require code that is generated from the synchronization specifications. With `SzumoSzep`, we are able to automatically generate this code and translate the subject program to use it, thus achieving our larger goal.

8 Related Work

Many others have worked on approaches, which automatically rewrite base programs to exploit synchronization based on some separated synchronization concern [18, 8, 20, 12, 7]. The work most closely related to ours is the D Framework, which uses an aspect-like language, called COOL, for expressing the coordination requirements of classes separately from their primary functionality [18]. COOL associates classes with *coordinators* in much the same way that Szumo uses synchronization units; however, coordinator specifications lack the local and compositional properties of our synchronization specifications. For example, each of the coordinator-based solutions to the dining philosophers problem must be modified to accommodate a change in the size of the configuration. That is, a coordinator specification is designed to work for a specific number of three philosophers and must be modified to accommodate more or fewer. In Szumo, the synchronization specification is unaffected by the size of the configuration.

Other related work focuses on the application of off-the-shelf AOP tools (e.g., AspectJ [16]) to achieve a separation of synchronization concerns. Rashid et al. used AspectJ to separate data persistence concerns [20], and Harbulot et al. experimented using AspectJ to separate performance (e.g., parallelism) and computational concerns in scientific computing systems [12]. While these approaches deal with the separation of synchronization concerns, the class of systems they apply to is much narrower than SzumoC++ making them difficult to compare. Cunha et al. implemented a collection of concurrency patterns in AspectJ [7]. However, as with any pattern-based approach, the burden of composition is placed on the developer.

Several approaches separate synchronization concerns for purposes of verification. Among these, the most closely related to our work is SyncGen, which employs a declarative specification of *region invariants* to generate synchronization code that is then woven into a subject program at predefined join points [8]. Region invariants represent an elegant mechanism for specifying synchronization in the style of conditional critical regions. They are not ideally suited for specifying synchronization constraints over sets of resources, such as are exemplified by the dining philosophers example. Also, SyncGen does not fully separate synchronization concerns from the functional code because there is no analog to an aspect-like language with pointcut designators. Rather, the programmer must identify join points in her code using using stylized comments.

Bultan and colleagues synthesize *concurrency controllers* from an operational action language specification [4]. Concurrency controllers implement global policies for sharing a resource while encapsulating the low-level synchronization logic required to implement these policies. The policies are expressed as collections of high-level guarded commands based on a set of predefined patterns. In addition to the policy specification, for every controller, the designer writes a *controller interface* dictating the acceptable sequences of calls the threads may make to the respective resources. By separating interface from implementation, concurrency controllers afford modular reasoning, but in a manner that is quite different from the compositionality of synchronization constraints in Szumo.

Magee and Kramer propose an articulate model-based methodology that separates synchronization and functional concerns for the purposes of verification and architectural design [19]. In this approach, the designer constructs and verifies an explicit model of the system. Once verified, elements of the model can be implemented idiomatically, though not automatically, as monitors in Java. This methodology separates concerns for the purpose of verification, however the modeling language is intentionally operational, and the overall approach does not address the code-tangling problem.

Vaziri, Tip, and Dolby proposed a declarative model of synchronization, which is similar in some respects to Szumo [26]. This model is implemented using Java language extensions whereby programmers declare sets of data that must be updated atomically. Their model is compositional and the compiler uses the atomic-set declarations to generate synchronization logic. Neither atomic sets nor Szumo synchronization constraints is strictly more expressive than the other.

9 Discussion and Future Work

In this paper, we presented a new approach to implementing strictly exclusive applications that separates synchronization and functional concerns. The approach extends our prior work on Szumo and combines several ideas, including:

- the use of an explicit design model, which has been verified and which informs the subsequent implementation of the functional and synchronization concerns,
- methods for achieving a high level of transparency between the design model and each of the implementation artifacts, and
- automated composition tools that exploit this transparency and dependence upon a shared design model.

In this section, we discuss some interesting corollaries of this work and our thoughts on future directions.

9.1 Explicit Design Modeling

Current best practices in multi-threaded program design make heavy use of models and modeling notations prior to (or in concert with) implementation [19]. The design of thread synchronization logic involves reasoning over state spaces that grow non-linearly with the size of the program. The virtue of models is that they are often able to represent the synchronization concern at a level of abstraction suitable for coping with the state-explosion problems that accompany property verification. Indeed, our Szumo design models are expressed at a high level of abstraction for precisely this reason [22, 9].

Unfortunately, most model-based approaches do not directly support the verification of conformance between the design models and their implementation. This makes the models prone to becoming outdated during the maintenance phase of a system's lifecycle in much the same way that documentation does. The problem is exacerbated when we then attempt to separate functional and synchronization concerns in the

implementation. Szumo design models are structured in a way that affords transparency with the implementation and thus simplifies the problem of conformance checking and also the composition of separately specified functional and synchronization concerns.

9.2 Design Transparency

We contend that to support the maintenance and evolution of large multi-threaded systems, design transparency is the key. This paper demonstrates how to retain transparency when a design is implemented using techniques for separating concerns. Achieving transparency in such an environment is not trivial, which is why our approach uses both OO frameworks and a heavy dose of generative programming. Specifically, we use an OO framework (`SimpleFrame`) and a custom synchronization specification language to impose a high level of design transparency on the implementations of both the functional and synchronization concerns.

Currently, we enforce design transparency through a combination of auditing procedures, whereby the developer manually checks his implementation against the design, and automated consistency checks that are performed by `SzumoSzep`. Checking the transparency of the functional logic involves verifying that each unit class inherits from either `Unit` or `RootUnit` depending upon the stereotype associated with the class in the unit-class model and that any inheritance relationships in the model are preserved in the code. Checking transparency of the synchronization logic involves checking that there exists a synchronization specification for each class in the unit-class model, and for each such specification checking that:

- there is a unit variable declaration for each association in the unit-class model,
- there is a condition variable declaration for each attribute in the unit-class model,
- with respect to a given condition variable, there is a transition clause for each transition in the synchronization-state model that affects that variable, and
- the synchronization constraints are imported directly from the unit-class model.

In future work, we intend to automate both of these auditing procedures in the context of an Eclipse-based UML development tool [11].

To further support our manual auditing procedures, we built into `SzumoSzep` the ability to detect some of the inconsistencies that might arise by separating the functional and synchronization concerns. For example, design transparency dictates that each unit class in a synchronization-optimistic program must have corresponding to a synchronization specification and vice versa. A developer could introduce an inconsistency by mistyping the name of the unit class in the synchronization specification. A fully automated technique for enforcing transparency would catch this problem prior to composition time. However, in absence of this capability, `SzumoSzep` detects and reports inconsistencies between synchronization specifications and their associated synchronization-optimistic programs.

9.3 Further Separation of Concerns

Anecdotal evidence suggests, and our experience affirms, that one cannot achieve a truly complete separation of synchronization and functional concerns. Rather, we opt for solutions that provide a clean separation and one that is robust under change. It is worth noting where, in our approach, there is the potential for tangling of synchronization and functional concerns. The obvious point is in the synchronization-specification language, where the condition variable triggering and cancelling is bound to pointcuts in the functional program. These pointcuts tend to be tightly coupled with the functional logic such that changes to this logic require re-examining and possibly modifying the pointcut designators in the synchronization specification. For example, a simple change to the name of the `Philosopher` method `eat` would result in the need to change the pointcut designator in the `Philosopher` synchronization specification. This problem is largely caused

by our use of syntax-level pointcut designators, which are also common to many general aspect-oriented programming tools (e.g., AspectC++ and AspectJ). There has been much interest in developing more expressive pointcut designators [20, 10, 13], with the apparent goal of achieving pointcut designators that are closer to the semantic level and would reduce the tangling described above.

We plan to expand and improve the expressiveness of the pointcut designators accepted by the synchronization aspect language. Initially, this would be primarily to increase the usability of our synchronization specifications; recall that at present, our pointcut designators can only express calls to methods with a particular name. An obvious improvement to our pointcut designators would be to enable distinguishing between calls by pattern-matching against the arguments, in addition to the name of the function. For example, pointcut descriptors of the forms

```
call(eat(*))
call(eat(x,y))
```

could be used to designate calls to `eat` with any combination of arguments, and arguments `x` and `y`, respectively. We also intend to implement pointcut descriptors for specifying assignments to member variables, which also incorporate pattern matching to improve their expressiveness. As we develop more expressive pointcut designators, we plan to increase our focus on exploring pointcut designators that are more semantic in nature, and thereby, less tightly coupled with functional code. However, an open question is the degree of power needed with regard to our pointcut designator language and consequent coupling of concerns. Moreover, there is also an open question of whether enabling pointcuts to refer to entities not represented in the design model will negatively impact design transparency in the the implementation.

9.4 Implementation Issues

That our synchronization specification language includes aspect-like concepts begs the question: Why did we use OpenC++ rather than an AOP tool, such as AspectC++ [24]. We now briefly describe our rationale and explain what we perceived to be the pros and cons of both technologies.

The factor that weighed most heavily on our decision was the need to check for inconsistencies between the synchronization-optimistic program and the synchronization specifications at composition time. OpenC++'s metaprogramming facilities support introspection, which provides a natural way to implement these checks. We were unable to find any support in AspectC++ for this type of base-program analysis. That said, we believe this deficiency could be addressed by enhancing AspectC++ with a feature such as AspectJ's `declare warning` or statically executable advice as proposed by Lieberherr et al. [17].

Another requirement of SzumoSzep that was straightforward to address in OpenC++, but was obscure in AspectC++, was the need to weave in interdependent advice. For example, consider a variant of the `Philosopher` class given earlier, whose `eat` method returns the number of times the philosopher has eaten so far. Assume the `eat` is called as follows.

```
int count = eat();
```

Now, consider the same statement in a synchronization-aware program produced by SzumoSzep based on the original `Philosopher` synchronization specification.

```
int count = ( eating = true ,
              notify_holder() ,
              tmp_var = eat() ,
              eating = false ,
              notify_holder() ,
              tmp_var );
```

Because the call to `eat` is embedded in an assignment expression, its return value must be stored in a temporary variable, `tmp_var`. This is done so that `eat`'s after advice (i.e., set `eating` to false and call `notify_holder`) can execute after the call to `eat`, but before the assignment expression is evaluated. The temporary variable is given at the end of the parenthetical expression, so it will be used by the assignment expression (as opposed to the value returned by `notify_holder`).

The need for interdependent advice arises because the temporary variable must be declared before it can be used; however this declaration cannot appear in before, after, or around advice to the call to `eat`.³ Essentially, the declaration involves another advice that must precede the statement in which `eat` was called (e.g., in our implementation, we place the declaration near the beginning of the enclosing method). We were unable to find a clean way to apply such interdependent advice in AspectC++. By contrast, OpenC++ enables the metaprogrammer to manipulate the parse tree representation of a method, which allows one to implement highly context-sensitive advice weaving, which is needed in this case to handle interdependent advice. We think this could be addressed in AspectC++ with a more expressive pointcut designator language.

Despite these benefits, OpenC++ was not without its own obscurities. Because SzumoSzep involves both the generation and weaving of code, the most natural way to organize the compiler involved translation in a sequence of consecutive stages. Unfortunately, the metaobject protocol provided by OpenC++ made it difficult to implement a truly staged translation of the base program, and our implementation of this is admittedly unclean. We considered trying to remedy the situation by reifying the stages as mixin layers [23], but our early experiments suggested that such advanced use of C++ templates would lead to a brittle and unmaintainable compiler. On the other hand, AspectC++ supports staged transformations at a much higher level through advice ordering. Using AspectC++, one can write an aspect for each type of translation and declaratively specify the order that aspects apply their advice to common join points. This functionality makes generating aspects from synchronization specifications much simpler than the technique we currently use for generating metaclasses that perform staged transformations.

9.5 Longer Term Future Work

As longer-term future work, we are investigating extensions or enhancements to Szumo to accommodate different categories of multi-threaded systems, i.e., categories that differ from the strictly exclusive. In particular, we would like to support systems that currently use read-write locks. To accomplish this will require a variant of `SzumoFrame` that (1) enables programmers to express whether an entailed unit is needed for writing or for reading only, and (2) to negotiate using new scheduling policies that distinguish between readers and writers. We are specifically interested in building frameworks that inculcate specific design decisions that simplify development using features that are difficult to use in general. So, for example, while there are known fairness issues inherent to the readers-writer problem in the abstract, implementations often make simplifying decisions, such as giving priority to writers in the interest of keeping data as current as possible. Thus a new category of systems might be those that exhibit readers-writer style sharing where the writers have priority.

Another interesting subset of the larger multi-threaded computing problem concerns high-performance computing applications, which exhibit a radically different synchronization profile than exclusive systems. An interesting question is whether a SzumoC++-like approach based on a different fundamental model of concurrency and synchronization would be feasible. To this end, we are looking at the abstractions provided in IBM's X10 language [5] among others.

Acknowledgements: Partial support for this research was provided by the Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9984726.

³because this assignment expression could be embedded within a larger expression.

References

- [1] R. Behrends. *Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages*. PhD thesis, Michigan State University, East Lansing, Michigan USA, December 2003.
- [2] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of FSE'2000*, 2000.
- [3] R. Behrends, R. E. K. Stirewalt, and L. K. Dillon. A self-organizing component model for the design of safe multi-threaded applications. In *Proc. of the ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'05)*, 2005.
- [4] Aysu Betin-Can and Tevfik Bultan. Verifiable concurrent programming using concurrency controllers. In *Automated Software Engineering*, 2004.
- [5] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. of the ACM 2005 OOPSLA conference*, October 2005.
- [6] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA*, 1995.
- [7] C. A. Cunha, J. L. Sobral, and M. P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.
- [8] X. Deng et al. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. of the IEEE International Conference on Software Engineering (ICSE'02)*, 2002.
- [9] L. K. Dillon, R. E. K. Stirewalt, B. Sarna-Starosta, and S. D. Fleming. Developing an Alloy framework akin to OO frameworks. In *Proc. of the First Alloy Workshop*, 2006. co-located with FSE'2006.
- [10] Rémi Douence, Thomas Fritz, Nicolas Lorient, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, 2005.
- [11] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, patterns, and plug-ins*. Addison-Wesley, 2004.
- [12] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004.
- [13] Matti Hiltunen, François Taïani, and Richard Schlichting. Reflections on aspects and configurable protocols. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.
- [14] Yutaka Ishikawa. MPC++ approach to parallel computing environment. *ACM SIGAPP Applied Computing Review*, 4(1), 1996.
- [15] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.
- [16] G. Kiczales et al. An overview of AspectJ. In *Proc. of the European Conference on Object-Oriented Programming*, 2001.
- [17] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003.
- [18] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997.

- [19] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 2000.
- [20] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003.
- [21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesley, second edition, 2004.
- [22] B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. In *Proc. of 18th Intl. Conf. on Softw. Eng. and Knowledge Eng.*, 2006.
- [23] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.
- [24] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proc. of the Fortieth International Conference on Tools Pacific*, 2002.
- [25] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared memory systems. In *Proc. of the 29th Annual IEEE/NASA Software Engineering Workshop*, 2005.
- [26] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Principles of Programming Languages*, 2006.

A Synchronization-Specification Language Grammar

Below we present the grammar for our synchronization-specification language. The grammar uses *italic* font for nonterminals and `typewriter` font for terminals. Note that braces, colons, commas and semicolons are all terminal symbols. Brackets are metasympols that group elements, and may be followed by one of two special metasympols. The first metasympol is a superscript ***, which indicates that the preceding group of elements may repeat zero or more times. The second metasympol is a subscript *opt*, which indicates that the preceding group of elements may occur zero or one times.

```
syncSpec      := syncSpecHead { syncSpecBody } ;
syncSpecHead := sync_spec className
syncSpecBody := [unitRefDecl]*
                [pointcutDecl]*
                [condVarDecl]*
                [constraintSpec]opt

unitRefDecl  := unit unitVar ;

pointcutDecl := sync_pointcut pcdVar : pcdExpr ;

condVarDecl  := condition condVar { condDeclBody }
condDeclBody := initDecl [adviceDecl]*
initDecl     := init(boolVal) ;
adviceDecl   := adviceType adviceTime : pcdExpr [, pcdExpr]* ;
adviceType   := trigger | cancel
adviceTime   := before | after

constraintSpec := constraint { constrBody }
constrBody     := [constrExpr ;]*

pcdExpr       := call(fun) | pcdVar

constrExpr    := true
                | unitVar
                | condVar ==> unitVar

boolVal       := 0
                | 1
                | true
                | false

className    := a legal C++ class name
unitVar      := a legal C++ variable name
condVar     := a legal C++ variable name
pcdVar      := a variable name
fun         := a legal C++ function name
```