Developing an Alloy Framework akin to OO Frameworks

L. K. Dillon, R. E. K. Stirewalt, B. Sarna-Starosta and S. D. Fleming
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA

E-mail: {ldillon,stire,bss,sdf}@msu.edu

Abstract

Object-oriented (OO) frameworks are known to provide tremendous benefits with respect to software reuse. Developers construct new applications through framework instantiation, which, in general, does not require understanding the implementation of the framework classes and methods. In prior work, we developed an OO middleware framework, called SzumoFrame, which supports the development and long-term maintenance of multi-threaded but strictly exclusive systems. While using Alloy to analyze models of programs that instantiate SzumoFrame, we discovered how to construct a reusable Alloy specification of SzumoFrame that can be customized with application-specific detail in a manner that is akin to the instantiation of an OO framework. The strong symmetry between this Alloy framework and SzumoFrame simplifies the construction of specifications and allows application developers to analyze candidate designs before committing to code.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*, *object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking, programming by contract*; D.2.13 [**Software Engineering**]: Reusable Software—*reuse models*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Alloy*

General Terms

Design, Languages, Verification

Alloy Workshop '06 Portland, OR USA

Keywords

Alloy, deadlock detection, design by contract, object-oriented frameworks, synchronization contracts

1. Introduction

Object-oriented (OO) frameworks provide tremendous benefits to software developers with respect to reuse. Algorithms and protocols that are reusable over many different applications can be abstracted and isolated into framework classes and framework methods. Doing so eases the construction of new applications, which only need to *instantiate* the framework. Instantiation involves filling in application-specific details, but generally does not require deep knowledge of how the reusable algorithm or protocol is implemented. This paper describes recent work on trying to apply ideas from OO frameworks to Alloy to simplify the generation and analysis of Alloy specifications of programs that are instances of an OO framework.

The work was performed in the context of a project to support component-based development of a class of multi-threaded OO applications. Termed *strictly exclusive systems*, this class comprises applications in which threads compete for exclusive access to dynamically changing sets of shared resources. Examples include extensible web servers and interactive applications with graphical user interfaces. 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 synchronization. Called Szumo[1], this model supports development and long-term maintenance of strictly exclusive systems that are robust under change [2, 3].

Szumo embodies the principles of design-by-contract, as articulated by Meyer [16]. It associates each thread with a *synchronization contract* that governs how the thread must synchronize with other threads. At run time, schedules for threads are derived by *negotiating* contracts on behalf of the threads. A thread is scheduled only once the thread's contract has been successfully negotiated. The contracts themselves are formed by conjoining module-level *synchronization constraints*, which a programmer declares in the modules' interfaces.

[1]for the SynchroniZation Units Model; an early version of this model was called the "universe model" [2]

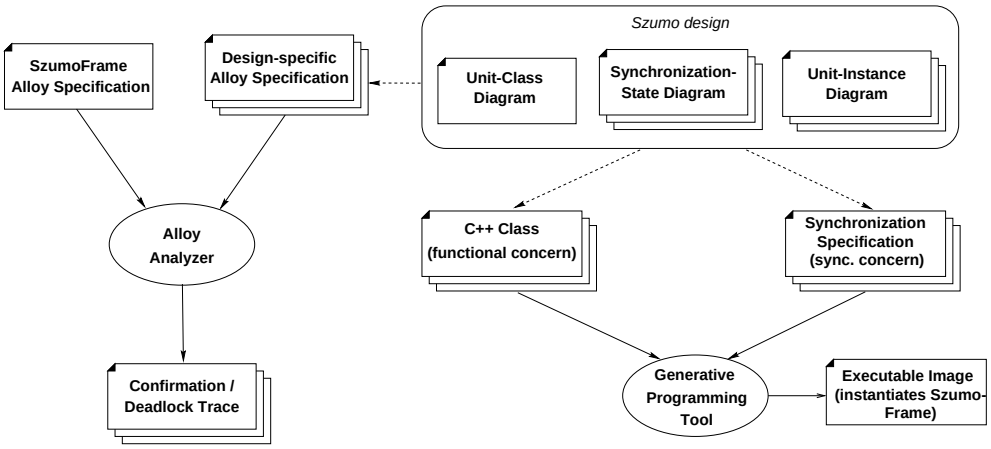

Figure 1. Overview of development approach

We recently implemented Szumo as an OO middleware framework, called SzumoFrame, and we are currently investigating the use of SzumoFrame for developing strictly exclusive systems. Figure 1 provides an overview of the approach, which affords a high degree of reuse through separation of functional and synchronization concerns, and provides strong correctness guarantees. The development starts with a Szumo design, which is used as a reference both prior to the implementation, to check for potential concurrency errors (e.g., race conditions or deadlock), and also during the implementation.

Our implementation strategy (Fig. 1, right side) exploits design transparency to support a clean separation of functional and synchronization concerns. An application developer first programs and debugs the functional logic (C++ classes), writing code that optimistically[2] assumes an executing thread has acquired exclusive access to any shared resources. She then separately programs the synchronization logic by writing declarative *synchronization specifications* and weaves the logic generated from these specifications with the functional logic using a generative-programming tool. The executable image generated by this tool instantiates SzumoFrame. In the figure, programming tasks are depicted using dashed arrows; whereas data flows into and out of automated tools are depicted using solid arrows. A declarative aspect-like notation is used for programming synchronization specifications. This notation provides high-level primitives that enable transparent specifications of Szumo designs. By virtue of this transparency, which must be discharged by checking that the C++ classes and synchronization specifications conform to the Szumo design, the application developer may focus her verification efforts on the Szumo design rather than the implementation.

When verifying Szumo designs, one important property to check is freedom from deadlock. The Szumo negotiation algorithm provides strong exclusion guarantees, while avoiding a large class of deadlocks. However, not all deadlocks can be avoided. We wanted to use the Alloy analyzer to check for potential deadlocks in Szumo designs; however, for an Alloy specification to accurately model the behavior of a Szumo design, it must include lots of detail regarding the semantics of negotiation. This is in sharp contrast with the implementation, which encapsulates the details of negotiation within the framework classes of SzumoFrame, thus simplifying the programming task.

This paper describes how we built an Alloy specification of SzumoFrame that can be customized with Szumo-design specific details to produce an Alloy specification of that design in a manner similar to how one would customize SzumoFrame (Fig. 1, left side). We refer to the Alloy specification of SzumoFrame as an *Alloy framework*. Our Alloy framework reifies the framework classes and methods of SzumoFrame as Alloy entities (atoms, relations, signatures, etc.). Just as SzumoFrame classes implement the details of dynamic contract negotiation, the analogous Alloy entities specify an abstract model of this negotiation. This strong symmetry simplifies the construction of models of Szumo designs and allows application developers to analyze various candidate designs before committing to code.

In the remainder of the paper, we provide necessary background on Szumo and SzumoFrame (Sec. 2). We then describe the Alloy entities that the application developer needs to know about in order to instantiate our Alloy specification of SzumoFrame (Sec. 3) and the process by which a developer instantiates it (Sec. 4). Finally, we provide some discussion (Sec. 5) and outline some related work (Sec. 6).

2. Background

We formulated Szumo to support development of strictly exclusive systems, for which a key problem is to synchronize threads that operate over shared data [2, 19]. Without proper synchronization, concurrent access to shared objects can lead to race conditions, and incorrect synchronization logic can lead to starvation or deadlock. Szumo is a language-independent model of synchronization contracts, which we integrated into an extension of the Eiffel language [1, 19] and, more recently, implemented in C++ as a framework, called SzumoFrame. This section supplies background on Szumo and SzumoFrame.

2.1 Szumo design concepts

[2] i.e., without worrying about synchronization,

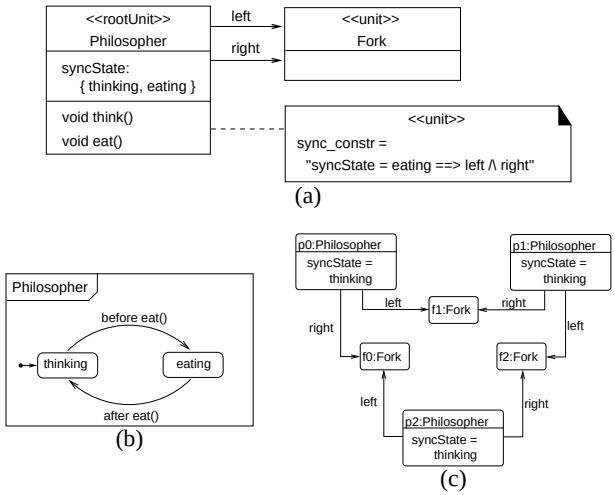

Figure 2. Unit-class diagram (a), synchronization-state diagram (b), and unit-instance diagram (c)

A fundamental design decision involves the granularity of sharing among threads. In Szumo, designers choose the granularity of sharing by deploying program objects into *synchronization units*, which are "object-like" containers of one or more program objects. When a program object is created, it is deployed to exactly one synchronization unit, where it remains throughout its lifetime. Threads are permitted exclusive access to objects at the level of synchronization units—that is, a thread that holds exclusive access to a synchronization unit holds exclusive access to all program objects contained within this unit. Because sharing occurs only at the level of units, concurrency analyses need not consider how units are composed of program objects.

Synchronization units are identified with instances of types, called *unit classes*. In addition to standard operations, a unit class may declare

- *Unit variables*
- A finite type together with a special attribute of this type, named `syncState`
- Synchronization constraints

A unit variable is a reference to another unit. As such, it indicates a direct client-supplier collaboration—the unit containing the variable acts as the *client* and the unit referenced by the variable acts as a direct *supplier*. The values that may be assigned to a unit's `syncState` attribute represent abstract *synchronization states*. We refer to a "unit's synchronization state" or to a unit as being "in a synchronization state" to indicate the value of the unit's `syncState` attribute. A synchronization constraint stipulates the synchronization states in which a client requires exclusive access to one or more direct suppliers. The synchronization constraint is said to be *triggered* when the client is in these states and to be *cancelled* when it is not. When a constraint is triggered the client is said to *entail* the stipulated direct suppliers.

To illustrate these ideas, Figure 2 shows a Szumo design for a solution to the classic dining philosophers problem. A *unit-class diagram* (Fig. 2 (a)) documents unit classes and relationships between them. We extend the UML class-diagram notation for this purpose using UML's built-in extension mechanisms. Stereotypes $\langle\langle$`unit`$\rangle\rangle$ and $\langle\langle$`rootUnit`$\rangle\rangle$ designate unit classes; the former indicates a unit class whose instances can be shared among multiple threads, and the latter a unit class whose instances serve as non-shared process "roots". In traditional design terms, sharable units correspond to passive objects, and root units correspond to active objects. We show the types of `syncState` attributes as sets and unit variables as directed associations. The synchronization constraints for a client class are given by the values for the "sync_constr" tags associated with that class. A synchronization constraint is formed using the entailment operator,"`==>`"; the first argument specifies the synchronization states that trigger the constraint and the second specifies a set of unit variables; the suppliers referenced by the unit variables in this set are entailed when the constraint is triggered. Thus, philosopher units execute in different threads and may perform operations on shared fork units which are bound to their `left` and `right` unit variables. The synchronization state of a philosopher is either `thinking` or `eating`. The synchronization constraint stipulates that, when in the `eating` state, the philosopher entails the fork units referenced by its left and right variables. This constraint is triggered when the philosopher's synchronization state is `eating` and cancelled when it is `thinking`; thus, a philosopher entails its left and right forks when it is `eating`. As no constraint is triggered in synchronization state `thinking`, a philosopher does not entail any forks when it is `thinking`. In contrast, fork units do not have synchronization states or synchronization constraints.

The *synchronization-state diagram* (Fig. 2 (b)) documents how operations that a unit may perform during execution affect its synchronization state. Synchronization states, shown as labeled rountangles in a unit's synchronization-state diagram, correspond to the synchronization states declared in its unit-class diagram. An arrow with no source state marks a unit's initial synchronization state, i.e., the synchronization state of a unit when the unit is first created. A unit changes its synchronization state as a result of *events* that occur during execution. Transitions, shown in the unit's synchronization-state diagram as arrows from a source state to a target state, specify the events, shown as labels on transitions, that cause a unit to change its synchronization state. A unit takes a transition when it is in the transition's source state and the event specified by the transition's label occurs; taking the transition leaves the unit in the transition's target state. Events are either *internal* to the unit or *joint* with one of the unit's direct suppliers. A label starting with either `before` or `after` and containing a method call is a joint event; `before` specifies that the event occurs immediately before a call on the method and `after` that the event occurs immediately after. All other events are internal to a unit. In our running example, a philosopher supplies its own `eat()` operation; the transitions are therefore joint between a philosopher and itself. When a philosopher is in state `thinking`, it takes the transition from `thinking` to `eating` immediately before a call to `eat()`; it then takes the transition back to `thinking` immediately after returning from the call. Together, the class diagram and the state diagram document that a philosopher entails its forks while executing an `eat` operation. As forks in this example have no synchronization states, the `Fork` class has no synchronization-state diagram. These intuitive definitions of events suffice for understanding the intent of a Szumo design. The true "meanings" of events actually depends on the synchronization specifications

(Fig. 1) that the developer creates to use in implementing the design.

Unit-class and synchronization-state diagrams are reusable over many different designs; whereas a *unit-instance diagram* (Fig. 2 (c)) depicts a particular configuration of synchronization units representing an initial state of a design. Although a design needs a minimum of five philosophers and five forks to exhibit behaviors in which multiple philosophers may be eating concurrently, for simplicity, we show a design with only three philosophers and three forks. A useful analysis strategy is to start with a simple unit-instance diagram, such as shown here, and analyze it first, before worrying about checking designs with more complex unit-instance diagrams.

2.2 Overview of the negotiation semantics

In a Szumo design, the set of units that a thread needs to access can be inferred at run time. For instance, from the design artifacts in Figure 2, 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 the smallest set of units that contains the units with activations on the thread's run-time stack and that is closed under the entails relation. The conjunction of the synchronization constraints associated with the units that a thread needs defines the thread's synchronization contract. When a thread executes an operation that changes the synchronization state of a needed unit or modifies the value of a unit variable in a needed unit, the units that the thread needs may change. A change in the needed units, in turn, may cause the thread's synchronization contract to change. SzumoFrame automates the negotiation of a thread's synchronization contract so as to satisfy the semantics below.

The semantics of negotiation are defined using a concept, called a *realm*. At run-time, each thread is associated with a realm, which is a set of synchronization units. A thread is allowed to access all and only units in its realm. Mutual exclusion is then guaranteed by requiring the realms of different threads to be disjoint. We say a thread *holds* a unit when the unit is in the thread's realm. Changes in a thread's contract may result in it 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 it is *damaged*. The semantics of Szumo dictate that a thread with a damaged realm must block until the realm is *repaired* (i.e., made complete). It may not be possible to immediately repair a damaged realm because some needed units may be held in the realms of other threads. A unit may be migrated into a realm only when it is not held by another realm or it is no longer needed by the thread that holds it. Finally, the migration of units into a damaged realm is atomic—all needed units not already held by the thread are migrated into the realm simultaneously.

From a user's perspective, a thread executes within a unit in its (complete) realm until it performs an operation that changes its contract, and thereby damages the realm. Consider, for instance, the dining philosophers configuration depicted in Figure 3 (a). The initial realm of each thread contains only the thread's root unit, a philosopher unit, each in its `thinking` state. The entailment of `p0`, `p1` and `p2` is the empty set and so these initial realms are complete. We depict complete realms as shaded rectangles surrounding the contained units. Before invoking the `eat()` operation, `p0`'s state changes to `eating`, thereby damaging the realm. Thus,

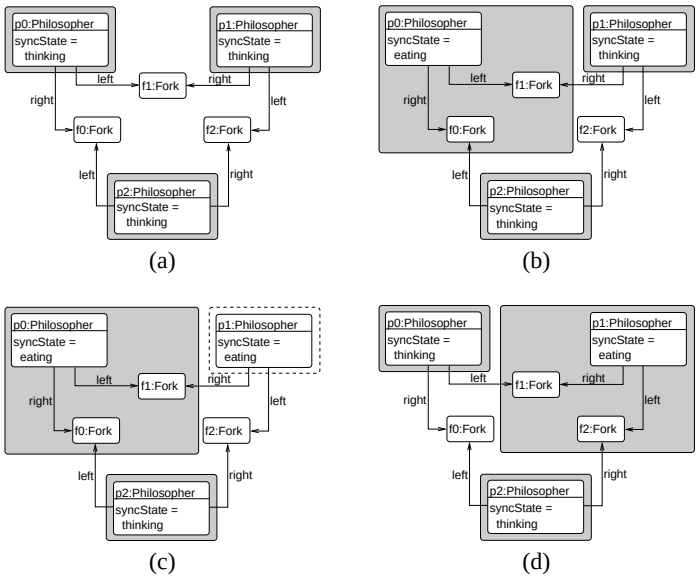

Figure 3. Snapshots of concurrency-relevant abstraction of a system execution

before the thread rooted in `p0` invokes `eat()`, its realm must expand to include the newly entailed forks `f0` and `f1`. As neither fork is held by other threads, they are atomically migrated into the realm, repairing the realm and producing the snapshot depicted in Figure 3(b). If at this point the thread executing in `p1` also changes its state to `eating`, the fork units `f1` and `f2` are added to `p1`'s entailment. Because `f1` is held by the thread rooted in `p0`, the realm containing `p1` cannot be repaired, and so the thread blocks. Although `f2` is not held by any thread, it is not migrated into the realm executing `p1` at this point, because the semantics dictate that the units needed to repair the realm must be migrated into the realm atomically. The thread blocks, therefore, while holding just the root unit `p1`. In Figure 3(c), the damaged realm is denoted by the dashed line. After the thread in `p0` returns from `eat()`, `p0`'s synchronization state changes to `thinking` in which the unit no longer entails `f0` and `f1`. This change affects the realms of both threads: the two fork units are released from the realm containing `p0`, and the realm of thread in `p1` expands to include `f1` and `f2` (Fig. 3 (d)).

In this simple example, it is easy to see that the design does not countenance deadlock. A philosopher can block when it takes the transition into its `eating` state, but not when it takes the reverse transition. Because a philosopher acquires the forks that it needs atomically, it either acquires both forks or neither fork. If it acquires both forks it does not block. Therefore, a philosopher blocks only if it is in its `eating` state and it does not hold any forks. But if none of the philosophers holds any forks, one of them can complete its realm. Hence, all philosophers cannot possibly be blocked. Moreover, the Szumo negotiation algorithm is also fair, guaranteeing that no philosopher that takes a transition into its eating state blocks permanently. In general, however, a Szumo design can countenance deadlock. Suppose, for example, that we designed the philosopher units using the unit-class and

synchronization-state diagrams shown in Figure 4 instead of those in Figure 2. A philosopher in this design executes internal events, which cause it to incrementally acquire and hold first its left and then its right forks, and then to release the forks in the reverse order. Clearly, this design is susceptible to the classic deadlock illustrated by the snapshot (Fig. 4 (c)).

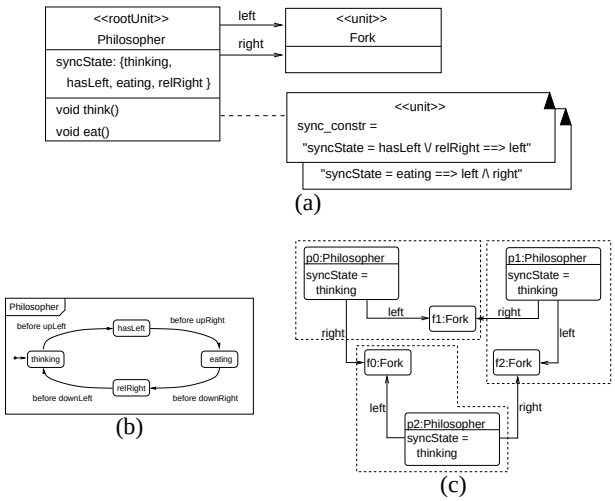

Figure 4. A Szumo design may countenance deadlock

2.3 SzumoFrame

An OO framework is an application skeleton that is fleshed into a concrete application by a process of instantiation, which involves two tasks—specialization and configuration. *Specialization* involves designing classes that extend one or more framework classes and adding new data or behavior at "hot spots"; whereas *configuration* involves writing code that allocates and configures instances of these new classes and/or instances of unextended framework classes, and typically a main program that cedes control to drivers provided by the framework. The framework classes, framework methods and drivers used in these steps constitute the framework's *instantiation interface*. To properly instantiate a framework, an application programmer needs to understand how the instantiation interface is meant to be used. However, the instantiation interface should be comparatively small and implement clean, easy to use abstractions. We designed SzumoFrame according to these principles.

SzumoFrame provides a framework class called `Unit` with which to define the unit classes of a given application. For example, the dining philosophers program contains two unit classes, `Philosopher` and `Fork`, both of which specialize the framework class `Unit`. Specialization involves three sub-tasks. First, the designer must declare an instance variable called `syncState` of a type that is appropriate for representing the various synchronization states of unit class instances. Second, she must supply a method called `entails`, which consults the `syncState` variable and returns a set of `Units` that represent the target unit's direct entailment. This method is invoked at run-time by the SzumoFrame code that is responsible for dynamic contract negotiation. Third, she must identify operations that could affect the synchronization relevant state of the unit. All such operations must be followed immediately by a call to the SzumoFrame method `damage_realm`, which initiates renegotiation of the calling thread's contract.

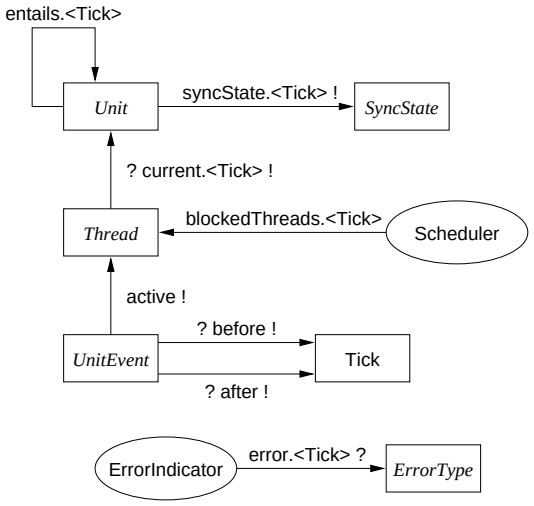

Figure 5. Domain model of the instantiation interface of our Alloy framework

The tasks performed to specialize class `Unit` derive from the unit-class diagrams depicted in Figure 2. Likewise, the configuration of instances of these specialized classes owes to the unit-instance diagram depicted in Figure 2. Because these design artifacts are the key to instantiating SzumoFrame, we also want to use them to instantiate our Alloy framework.

3. The Alloy framework's instantiation interface

The instantiation interface of our Alloy framework comprises a set of signatures and relations, which will be extended and configured; a set of functions and predicates, which will be invoked or defined; and a set of assertions, which will play the role of "drivers" during framework instantiation. Figure 5 depicts the central signatures and relations in the instantiation interface as a *model diagram* [12]. Briefly, a rectangle represents an arbitrary signature, an oval represents a signature containing just one atom, and an arrow represents a relation. To signify an abstract signature, the label on a rectangle is italicized. The direction of an arrow reflects the ordering of a relation's tuples; in the case of a binary relation, the relation's domain is contained in the signature at the source end and its range is contained in the signature at the target end. The label on an arrow is either a relation name or *archetypal expression* and, in either case, it may include multiplicity symbols on either or both ends—a multiplicity symbol preceding a relation name or archetypal expression signifies the multiplicity at the source end and one following a relation name or archetypal expression signifies the multiplicity at the target end. The multiplicity symbols are:

*	any (the default)	!	exactly one
?	zero or one	+	one or more

In the absence of a multiplicity, the most general (any) is assumed. In this paper, all archetypal expressions are of the form

rname.<*SigName*>

where *rname* and *SigName* denote, respectively, the names of a ternary relation and of a signature whose atoms are totally ordered. An arrow labeled with such an expression represents a sequence of relations indexed by *SigName*—more precisely, for every atom s in *SigName*, the expression $rname.s$ relates the source and target signatures and has the indicated multiplicities. In Figure 5, for example, `syncState.s` relates each unit to exactly one synchronization state, for `s:Tick`.

Our framework encapsulates a variation of an Alloy idiom that uses a totally ordered set of atoms to represent the points in time at which events occur in a (finite) execution of a state machine. Our instantiation interface therefore provides a signature,`Tick`,[3] to represent points in time and a signature, *UnitEvent*, to represent events. The latter signature is abstract. Henceforth, we refer to atoms belonging to these signatures more informally as just *ticks* and *events*, and to atoms belonging to other signatures in a similarly informal fashion.

In keeping with the Alloy idiom, `before` relates an event to the tick at which the event occurs, while `after` relates the event to the next tick, which will be either the tick at which the next event occurs or, if there are no further events, the last tick. Both relations are partial functions, reflecting our decision to model events as instantaneous and to simulate concurrency by interleaving.

When modeling Szumo designs, an event represents a transition taken by a thread while executing a unit. This thread is said to be *active* at the tick when the event occurs. The relation `active` represents the association between events and their active threads—more precisely, for `e:UnitEvent`, the expression `e.active` denotes the thread that is active at tick `e.before`.

The unit defining the transition that a thread executes is the unit at the top of the thread's stack. We refer to this unit as the thread's *current unit*. To model the mapping of threads to current units over time, we define the ternary relation `current`. In essence, this relation represents a sequence of assignments, ordered by ticks. For `t:Thread` and `s:Tick`, the expression `current.s[t]` denotes the unit that thread `t` will execute at tick `s`, assuming that thread `t` is active at tick `s`.

The abstract signatures *Unit* and *SyncState* represent synchronization units and synchronization states, respectively. The expression `entails.s`, relates units to the units that they entail at tick `s`; similarly, `syncState.s` relates units to their synchronization states at tick `s`, for `s:Tick`.

To facilitate detection and visualization of executions that end in deadlock, our Alloy framework makes blocking visible. We model blocking using a singleton `Scheduler` and the `blockedThreads` relation. For `s:Tick`,

`blockedThreads.s[Scheduler]`

denotes the set of threads that are blocked at tick `s`. We also introduce a sole `ErrorIndicator`, an `ErrorType` signature, and the `error` relation. The `error` relation is empty until an error occurs, after which point it relates the `ErrorIndicator` to an atom indicating the type of error that occurred. While not strictly a part of the instantiation interface, these signatures and relations are useful for defining new analyses.

In addition to the signatures and relations in Figure 5, the instantiation interface of our Alloy framework includes predicates needed to express design-specific details of an initial design configuration and of events, as well as assertions and error signatures to use in defining commands to perform analyses. We summarize these remaining entities in Table 1.

To instantiate the Alloy framework, the developer will extend the abstract signatures in Figure 5; introduce invariants to represent design-specific details of units, synchronization states, threads, events, and the `entails` relation; and define the first two initialization predicates in Table 1. Thus, the abstract signatures, the `entails` relation, and the two initialization predicates are essentially hot spots, where design-specific details must be supplied during instantiation. The framework defines the other entities in Figure 5 and Table 1 to be used in specifying the design-specific details.

Hidden underneath this small instantiation interface are Alloy definitions that encode the semantics of negotiation in Szumo at the level of detail needed to detect deadlocks. The next section explains how the Alloy framework is instantiated to produce an Alloy specification from a Szumo design, together with Alloy commands to carry out an analysis.

4. Instantiating the Alloy framework

Similarly to instantiation of SzumoFrame, instantiation of our Alloy framework involves both specialization and configuration. Table 2 describes the steps for specializing a unit class C in a Szumo design. Figure 6 illustrates these steps, showing how we specialize the design in Figure 2.

Referring to the unit-class model in Figure 2, we first introduce signatures to represent the synchronization states of philosophers [Fig. 6, lines 4 and 6]. Second, we introduce philosopher and fork units as extensions of `Unit` and supply invariants to define their `entails` relations [lines 1–2 and 8–15]. In this step, we also introduced relations `left` and `right` for the unit variables declared in a philosopher class. Because the unit variables in this design are immutable, we represent them as binary relations; if they had been mutable, we would have instead used ternary relations. The invariant for a fork [line 2] asserts that the fork never entails any units, reflecting the fact that the design does not associate synchronization states with forks. The invariant for a philosopher [lines 13–15] asserts that, at every tick, either the philosopher's synchronization state is `eating` and it entails the units bound to its left and right variables, or its synchronization state is `thinking` and it does not entail any units. Thus, this invariant reflects the semantics of the synchronization-state diagram and the synchronization constraint for philosophers (Fig. 2). In the third specialization step, we produce signatures and invariants for events that represent transitions in a philosopher unit. The first of these is an abstract signature, which all events representing transitions in a philosopher will extend [line 17–18]. Its invariant requires that the unit at the top of the active thread's stack is a philosopher unit. We define two types of events, corresponding to the two transitions in a philosopher's synchronization-state model—`BeforeEat`-events [lines 20–27] and `AfterEat`-events [lines 29–36]. In both invariants, `cu` denotes the philosopher unit at the

[3] called `Time` in [12, Section 6.2.4]

Initialization predicates:

- `initializedUnits(s:Tick)`
  Must be defined during instantiation; asserts that all units are in their initial synchronization states at tick `s`
- `initializedThreads(s:Tick)`
  Must be defined during instantiation; asserts that all threads are initialized with the appropriate root units at tick `s`
- `initThread(t:Thread,r:Unit,s:Tick)`
  Defined in the framework; asserts that thread `t` is initialized with root unit `r` at tick `s`

Predicates to use in defining events

- `isInvokeEvent(t:Thread,u:Unit, s,s′:Tick)`
  Defined in the framework; asserts that the event represents an invocation of a method in unit `u`, that `t` is the active thread, and that the event's before and after ticks are `s` and `s′`, respectively.
- `isReturnEvent(t:Thread,s,s′:Tick)`
  Defined in the framework; asserts that the event represents the return from a method invocation, that `t` is the active thread, and that the event's before and after ticks are `s` and `s′`, respectively.
- `isInternalEvent(t:Thread,s,s′:Tick)`
  Defined in the framework; asserts that the event involves just the current unit, that `t` is the active thread, and that the event's before and after ticks are `s` and `s′`, respectively.

Error signatures (defined in the framework)

- `DeadlockError`: Indicates all threads are blocked.
- `DeadCycleError`: Indicates the existence of a set of two or more threads all of which are blocked and each of which holds a unit needed by another
- `StackError`: Indicates a stack overflow or underflow.

Table 1. Additional functions, error signatures and assertions in the instantiation interface

1. If there is a synchronization-state diagram for C,
   (a) extend *SyncState* with an abstract signature representing the synchronization states of C, and
   (b) for each synchronization state, extend this latter signature with a singleton signature representing the state.
2. Extend *Unit* with a signature representing units of type C and containing:
   (a) for each unit variable in C, a relation representing the binding of the variable to a unit, and
   (b) an invariant defining C's `entailment`
3. If there is a synchronization-state diagram for C,
   (a) extend *UnitEvent* with an abstract signature representing events executed by units of type C and
   (b) for each transition in the synchronization-state diagram, extend this latter signature with a signature representing events in which a unit of type C takes the transition.

Table 2. Specialization of a unit-class C

```
1  abstract sig Fork extends Unit {}
2  { no entails }
3
4  abstract sig PhilSyncState extends SyncState { }
5
6  one sig eating, thinking  extends PhilSyncState{ }
7
8  abstract sig Philosopher extends Unit  {
9    left, right: Fork
10  }
11 {
12   all s: Tick {
13     (eating=syncState.s  &&  entails.s=left+right)
14     ||
15     (thinking = syncState.s  &&  no entails.s) } }
16
17 abstract sig PhilEvent extends UnitEvent { }
18 {current(active, before) in Philosopher}
19
20 sig BeforeEat extends PhilEvent { }
21 {
22   let cu=current(active, before) {
23     cu.syncState.before = thinking
24     cu.syncState.after = eating
25     isInvokeEvent (active, cu, before, after)
26    }
27  }
28
29 sig AfterEat extends PhilEvent{ }
30 {
31   let cu=current(active, before) {
32     cu.syncState.before = eating
33     cu.syncState.after = thinking
34    }
35   isReturnEvent(active, before, after)
36  }
```

Figure 6. Example of specialization

top of the active thread's stack—in other words, the philosopher unit taking the transition represented by the event. The invariant associated with an event supplies the following details: the synchronization state of `cu` before the transition, the synchronization state of `cu` after the transition, and whether the transition corresponds to the invocation of a method, the return from a method invocation, or an event that is internal to `cu`. The definition of an event is the analog of invoking the `damage_realm` method when instantiating SzumoFrame. It makes the effects of a transition visible, thus triggering the negotiation machinery, which is hidden under the instantiation interface.

1. Extend *Thread* and the unit signatures defined during specialization with singleton signatures representing the threads and units in the initial unit-instance model
2. Introduce an invariant expressing the assignment of units to unit variables in the initial unit-instance model
3. Define the initialization predicates for units and threads
4. Express the analyses to be performed as Alloy commands

Table 3. Configuration

```
1  one sig T0, T1, T2 extends Thread { }
2  one sig F0, F1, F2 extends Fork { }
3  one sig P0, P1, P2 extends Philosopher { }
4
5  fact UnitConfiguration {
6    P0.left = F0 && P0.right = F1 && P1.left = F1
7    P1.right = F2 && P2.left = F2 && P2.right = F0
8   }
9
10 // initialization predicates
11 pred initializedUnits (s: Tick) {
12   Philosopher.syncState.s =  thinking
13  }
14
15 pred initializedThreads(s: Tick) {
16   initThread(T0, P0, s) && initThread(T1, P1, s) && ...
17  }
18
19 check noDeadlock for 6 Tick, 3 Sindex, 5 Event
20 check noError for 10 Tick, 3 Sindex, 9 Event
21 check noError for 10 Tick, 1 Sindex, 9 Event
```

Figure 7. Example configuration

Table 3 lists the steps involved in configuring a Szumo design. Figure 7 illustrates the application of these steps. Referring to the unit-instance model in Figure 2, in the first configuration step, we produce signatures for three threads, three forks, and three philosophers [Fig. 7, lines 1–3] and, in the second, we introduce a fact defining the bindings of unit variables to forks [lines 5–8]. Similarly, referring to the synchronization-states model in Figure 2, we introduce initialization predicates for units [Fig. 7, lines 11–13] and threads [lines 15–17], in the third configuration step. In defining the latter initialization predicate, we use predicates defined in the framework to associate a root unit with a thread. Finally, in the last step, we define the analyses to be performed. Each command must specify a maximum number of ticks and events in the execution traces that are to be checked, as well as a maximum stack size. The number of events should always be one less than the number of ticks. The first command checks whether any traces of length five or less[4] and with a maximum stack size of three can result in deadlock [line 19]. The second and third commands [lines 20–21] check for errors corresponding to any of the error types defined in the framework. They check traces of length nine or less, but with different maximum stack sizes. These commands are the analogs of different "main programs" that might be used with an OO framework. The commands essentially cede control to assertions defined in the Alloy framework. Thus, these assertions are analogs of drivers in an OO framework.

5. Discussion

This paper introduces the notion of an Alloy framework, i.e., a collection of entities and relations that model a reusable infrastructure, which must be customized to yield an application. We developed the idea while constructing Alloy models of Szumo programs, which are constructed by instantiating an OO framework called SzumoFrame. OO frameworks allow programmers to quickly develop programs that reuse complex infrastructures by extending framework classes at hot spots, configuring instances of the extended classes, and ceding program control to drivers. Our results suggest that Alloy specifications can be organized to afford a similar style of reuse at the level of models of programs that reuse complex infrastructures.

It is difficult to quantify the "complexity" of the reusable Alloy specification in order to compare it with non-reusable specification, which the developer writes. Our Alloy framework contains 10 non-trivial signatures and 8 relations. We introduce 9 functions and 17 predicates to simplify specification of a fairly complex invariant. Our final specification of SzumoFrame comprises 174 non-comment, non-blank lines of Alloy code. These measures are admittedly arbitrary (e.g., the number of lines of code depends on formatting style, use of functions and predicates, etc.). However, these measures indicate that the framework specification is significantly more complex than the specification that the developer writes. Moreover, as demonstrated in Section 4, this latter specification is routine to write.

We ran Alloy commands like those in Figure 7 (but with larger scopes) on Alloy specifications produced from both versions of the dining philosophers designs in Section 2 (and with different numbers of philosophers and forks). As expected, no deadlocks were found in the first design and the classic deadlock was found in the second one when checking sufficiently long traces. For the first design, the command to check for errors produced a stack error when the stack size was set to a maximum of one. The second design does not exhibit this error because all events modeled by the design are internal events. We now briefly discuss some of the open questions raised by this work as well as some of the larger issues in the design and verification of high-assurance systems whose implementations leverage an OO framework.

To date, we have investigated analysis for only deadlock, dead cycles, and stack errors (Table 1). We have to check for the latter in order that deadlocks and dead cycles are not quietly masked

[4]The framework encapsulates a special "no-op" pseudo-event, which it uses to fill out execution traces that end prematurely.

by too small of a stack size. There are many other interesting properties that a developer might like to check for a given Szumo design. Many will be design-specific. An interesting challenge for this work, therefore, will be to extend the Alloy framework to simplify checking of other propertis and, in particular, of design-specific properties. For instance, it might be useful to extend our Alloy framework with support for creating Alloy encodings of common temporal specification patterns [7]. Pure liveness cannot be checked on finite traces, but many of these patterns express safety and bounded liveness, which could be checked using the Alloy analyzer.

An important open question is whether models constructed by instantiating an Alloy framework suffer a performance penalty as compared to hand-built models. The need to combat state explosion imposes hard limits on the extent to which a designer may trade additional state to simplify extension and reuse. A developer extends our Alloy framework by imposing invariants and by specifying configurations. We expect that invariants would tend to limit, rather than contribute, to state explosion and that configurations would need to be specified even in hand-built models. The structures in our Alloy framework are introduced because they represent important aspects of the phenomena to be analyzed, not merely because they simplify the construction of models. That the structures do simplify the construction of models derives from a clean separation of concerns in the design of SzumoFrame. We believe that other uses of OO frameworks may lend to similarly competitive Alloy frameworks provided the OO framework affords a clean separation of concerns.

Another issue concerns the types of analysis obligations that we may reasonably discharge using Alloy in the environment depicted in Figure 1. If the tool reports a deadlock, then there is a flaw in the design. Because the Alloy model is transparent with respect to the design, the counter-example should prove useful for tracing the flaw. However, we cannot interpret a "no deadlock" result to mean that a design does not countenance deadlock. An open question is what to do if Alloy reports no deadlock. We are also considering other analysis tools. For example, in [18], we show how to use constraint logic programming (CLP) to analyze Szumo design artifacts. Analysis of the CLP model is performed on-the-fly, without imposing a maximum analysis depth; it can therefore uncover deadlocks that the Alloy analysis cannot find within the specified scopes. We are still assessing the relative merits of using Alloy or CLP. Our current encoding of designs using CLP is less transparent than the encoding in Alloy, which is described in this paper. While much of the CLP encoding is reusable, we have yet to explore whether we can separate the reusable portions into a "framework" as cleanly as we were able to do with Alloy. Preliminary experiments indicate that analysis of CLP design models may scale better than analysis of Alloy design models.

An extended Alloy system, called DynAlloy [10], might provide a performance advantage and also simplify the specification of SzumoFrame. DynAlloy extends Alloy with *actions* and Hoare-style partial correctness assertions, without requiring explicit specification of traces. It uses a modified version of the Alloy analyzer. Empirical results indicate that the modified analyzer permits more efficient checking of partial correctness.

In Figure 1, the developer analyzes a Szumo design and refines it into an executable system in two separate processes. However, for the results of analysis to be useful in reasoning about the correctness of the refined system, the implementation must conform to the design in some verifiable way. We believe it should often be possible to check whether the implementation of a unit class conforms to the Szumo design models by a data-flow analysis that propagates the synchronization states of units. Future work will look at using interprocedural data flow analysis [11] and recent work on statically checking conformance of code with typestates [8] to automate the verification of conformance between Szumo design models and code.

Finally, we continue to extend and explore the engineering benefits of Szumo. Szumo, itself, is fairly mature. We have integrated support for it into programming environments for Eiffel and for C++. We used Szumo Eiffel in a substantial case study to corroborate the benefits of synchronization contracts during maintenance of a realistic multi-threaded system [3]. Our newest integration, called SzumoC++, incorporates ideas from aspect-oriented programming [14] to further separate functional and synchronization concerns. The environment depicted in Figure 1 reflects the architecture of SzumoC++.

6. Related work

Numerous groups have investigated techniques for modeling and formalizing OO frameworks. One area of work focuses on informal modeling for the purpose of design specification. Kobryn [15] and Zhu and Tsai [21] focus on methods for producing quality designs for systems that incorporate OO frameworks. Fontoura et al. [9] present a method for producing framework-based designs, which they further support with a modeling language, UML-F, which extends UML to enable the explicit representation of framework variation points. Nakajima and Tamai [17] developed a method by which they were able to perform automated behavioral analysis of the Enterprise JavaBeans component architecture using the SPIN model checker. The work most closely related to ours is that of Chen [5], which proposes to construct a formal model of an OO framework. Here, the idea is to use Java Path Finder (JPF) to analyze code that instantiates the framework. Their work appears to be motivated by the need to apply abstractions and heuristics when searching the state space of a program that would otherwise be too large to analyze were JPF to explore the code for the framework classes.

Others have also found that Alloy provides a useful basis for analyzing properties of program designs. For example, in [6], Alloy is used to perform commutativity analysis at the level of designs. In a multi-user system, non-commuting operations that are issued simultaneously by different users can produce unexpected outcomes. The paper shows how to translate OCL specifications of operations into Alloy predicates and then, from the predicates for a pair of operations, how to build an Alloy assertion requiring that the operations commute. The Alloy analyzer is used to check for counterexamples, which represent scenarios for which the operations do not commute. In [13], Alloy is used to model a design of a new scheme for resource discovery in a dynamically evolving network. Analysis of the model reveals several flaws in the design. In [4], Chang and Jackson build models of OO programs in a notation that mixes relational operators, primitive data types and standard imperative features. They implemented a BDD-based model checker for this notation, which they use in verifying properties expressed in CTL. In one example, they analyze a mutual exclusion algorithm, showing that mutual exclusion is always satisfied and that the algorithm does not countenance deadlock. They express dependencies among processes and resources in terms of

`has` and `wait` relations. We use similar relations in encoding the semantics of Szumo negotiation. Finally, a type of Alloy framework is used in [20] to model and verify correctness of multicast key management schemes. This Alloy framework is reusable, but over a much narrower range of applications than ours. To the best of our knowledge, no other work facilitates design transparency and reuse of an extensible Alloy model to the extent achieved using our Alloy framework.

7. Acknowledgments

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.

8. 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, Dec. 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] F. S.-H. Chang and D. Jackson. Symbolic model checking of declarative relational models. In *ICSE'06: Proc. of the 28th International Conference on Software Engineering*, pages 312–320, 2006.

[5] Z. Chen. Formal modeling: A framework-based approach. to be presented at the FSE'06 Doctoral Symposium.

[6] G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *ISSTA'04: Proc. of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 165–174, 2004.

[7] M. Dwyer, G. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the* 21^{st} *Int'l Conf. on Software Engineering*, 1999.

[8] S. Fink et al. Effective typestate verification in the presence of aliasing. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'06)*, 2006.

[9] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A modeling language for object-oriented frameworks. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, 2000.

[10] M. Frias, J. P. Galeotti, C. Lopez Pombo, and N. Aguirre. DynAlloy: upgrading Alloy with actions. In G.-C. Roman, editor, *ICSE 2005: 27th International Conference on Software Engineering*, pages 442–450, New York, NY, USA, 2005. ACM Press.

[11] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proc.* 3^{rd} *ACM SIGSOFT Symp. on Softw. Testing, Analysis, and Verification*, pages 158–167, Dec. 1989.

[12] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[13] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *ASE'00: Proc. of the 15th IEEE International Conference on Automated Software Engineering*, pages 13–22, 2000.

[14] G. Kiczales et al. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[15] C. Kobryn. Modeling components and frameworks with UML. *Commun. ACM*, 43(10), 2000.

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[17] S. Nakajima and T. Tamai. Behavioural analysis of the enterprise JavaBeans component architecture. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, 2001.

[18] 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* 18^{th} *Intl. Conf. on Softw. Eng. and Knowledge Eng.*, 2006.

[19] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared memory sytems. In *Proc. of the* 29^{th} *Annual IEEE/NASA Software Engineering Workshop*, 2005.

[20] M. Taghdiri and D. Jackson. Lightweight modelling and automatic analysis of multicast key management schemes. In *FORTE'03: Proc. of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, pages 240–256, October 2003.

[21] F. Zhu and W.-T. Tsai. Framework-oriented analysis. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, 1998.