Using views to specify a synchronization aspect for object-oriented languages

R. E. K. Stirewalt, L. K. Dillon, R. Behrends
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{*stire,ldillon*}*@cse.msu.edu*

Abstract

It is widely held that programming language extensions that support separation of concerns and that are also integrative benefit development, maintenance and reuse of software designs and code. Such is the intent of our Synchronization Units Model (Szumo), which unifies new features for expressing synchronization in a multi-threaded program with existing features of an object-oriented language. However, to make effective use of a language extension, a programmer needs an accurate mental model of how new concepts affect and are affected by existing concepts. Moreover, good separation dictates that interactions between these concepts should be understandable at the level of the new concepts. This suggests that the semantics of Szumo should be specifiable as a self-contained partial specification, called a view, and the semantics of its integration with other language features should be specifiable by view composition. To our knowledge, however, view-based approaches have not been applied in specifying the semantics of language extensions. Moreover, devising separable views that serve to simplify comprehensibility of a complex specification is still more of an art than a science. This paper presents a case study in the use of views in structuring a Z specification of Szumo.

1. Introduction

The latest research developments in programming languages support improved *separation of concerns* by extending OO programming languages with new language features—e.g., aspects [17], traversal strategies [19], and typestates [9]. These new features provide powerful abstractions with which to express concerns that are difficult to localize in a single class—e.g., logging, assertion checking, and traversals of linked object structures. Such extensions often incorporate special-purpose programming paradigms—e.g., logic formulas and regular expressions—that facilitate expression of some concern as a *program aspect*. It is widely held that such separation of concerns benefits development, maintenance and reuse of complex software designs and code.

One of the most challenging concerns to separate is the logic for synchronizing threads in a concurrent program. Proper synchronization often requires the development of complex *negotiation protocols* through which multiple threads cooperate to safely access shared resources while avoiding starvation and deadlock. In addition to being difficult to design correctly, implementations of these protocols tend to be highly interleaved with the "functional core" of a program. To separate these concerns, researchers have developed custom aspect languages, such as COOL [20], for programming *synchronization aspects* separately and then weaving them into the functional core. Unfortunately, while the negotiation logic may be physically separated into an aspect, it is often not possible to reason about this aspect in isolation or about its interaction with the functional core [8].

To address this deficiency, we developed a linguistic model for separating synchronization from functional concerns in a manner that facilitates reasoning about their composition. Our *synchronization units model (Szumo)* [4, 3, 27][1] extends object-oriented languages with declarative features for expressing the synchronization concern. Using Szumo, the programmer writes code to allocate processes[2]; however, in lieu of operational code to synchronize the processes, he declares how his objects cluster into *synchronization units* and associates a *synchronization constraint* with each unit. A synchronization constraint specifies a unit's access needs (i.e., those units that objects within the given

[1]called the Universe model in early publications.
[2]or threads—throughout the paper we treat "processes" and "threads" as synonyms.

unit access directly) as a function of the unit's local state.[3] At runtime, processes negotiate with one another for exclusive access to units in order to satisfy these constraints, all while avoiding starvation and a large class of deadlocks. Automated negotiation frees the programmer from having to implement many of the details of synchronization.

To take advantage of Szumo features, a programmer does not need to understand the implementation of the negotiation machinery. However, she does need to understand how Szumo concepts (e.g., units and synchronization constraints) and operations (such as changes to synchronization states) affect and are affected by the concepts and operations of object-oriented programs. This suggests that synchronization aspects in Szumo should be specifiable as a self-contained partial specification, known as a *view*, and that the integration of these aspects with the functional core should be specifiable by *view composition*. Views afford a powerful separation of concern. They allow complex specifications to be constructed incrementally by composing simpler partial specifications, each of which describes some aspect of the full specification. Their use is recommended in structuring specifications of complex systems for comprehensibility [1, 6, 10, 16, 29].

This paper contributes a view-based specification of the semantics of Szumo synchronization aspects and their integration into multi-threaded, but synchronization-unaware, object-oriented programs. Using the Z notation [26], we define a $BaseView$, which models the synchronization-unaware programs, and a *SynchView*, which models the concepts and operations related to process synchronization. The synchronization view is understandable in its own right. In composing the base-language and synchronization views, inter-view invariants and joined operations define the effects of each view on the other. The result is a view-based semantics of Szumo as an extension of object-oriented languages. We are not aware of views having been used to specify the semantics of language extensions in this manner.

A secondary result is a set of conventions for structuring the two views to simplify their separate specification and later composition. In addition to being useful for composing these two views, we were able to apply these conventions recursively to decompose the specification of the synchronization view as an orderly composition of smaller views, each of which formalizes some Szumo concept in isolation of the others. Page limitations prevent the depiction of the full semantics, which can be found in [28]. Rather, we illustrate key aspects of the model that lend insight into the virtues that derive from the use of views and some of the lessons we learned in doing so.

[3]The "synchronization relevant" local state of a unit is represented explicitly via a collection of boolean *condition variables*. Allocation of processes and assignments to condition variables are the only imperative features of Szumo; the latter are currently unified with assignments to boolean valued instance variables of designated objects, but a new version of Szumo (under development) will abstract them into declarative conditions, which could be affected by general operations in a program.

2. Background on Szumo

We invented Szumo to separate the specification of synchronization from the specification of a program's core functionality. In a Szumo-extended programming language, synchronization is specified in a declarative constraint language, and the core computation is programmed in the OO paradigm using the base (unextended) language. We formalize the semantics of a Szumo-extended program as the composition of two sub-programs: the base OO program and a *synchronization program*, which is an abstraction of the processes and the objects in the base program and whose execution is governed by the synchronization constraints. We now informally introduce the concepts and operations of synchronization programs, irrespective of their connection to the base program.

In a synchronization program, synchronization units model cohesive clusters of base-program objects with similar synchronization needs, and processes operate in disjoint *realms* that comprise one or more synchronization units. A synchronization unit may reference other units to model the existence of a client–supplier relationship in the base program. Each client unit is annotated by a synchronization constraint that specifies the conditions under which the client requires exclusive access to its direct suppliers. As a synchronization program executes, its realms dynamically reconfigure to schedule processes for execution while ensuring that 1) realms are always pairwise disjoint, 2) a process is scheduled only when its realm consists of a distinguished *root unit* and all suppliers required by synchronization constraints associated with units in the realm, and 3) starvation and preventable deadlocks are avoided.[4] Because realms are disjoint, two processes are assured to never concurrently access the same memory location—shared units migrate from one realm to another, but these units are never accessed simultaneously by different processes.

Figure 1 illustrates two snapshots in the execution of a synchronization program that models a solution to the familiar dining philosophers problem. This program comprises two processes that execute over a total of four synchronization units, which are uniquely named for ease of reference. The `Philosopher` units (p_1 and p_2) run in their own processes, whose realms are shown using dashed or dotted boxes that enclose the units contained in the realms. For example, in **Stage 0** of **Figure 1**, the realms of the two processes contain only their root units (i.e., p_1 and p_2). No process may access either of the two `Fork`

[4]Our reference implementation of Szumo (as an extension to the Eiffel language) includes a run-time system that implements these features [3].

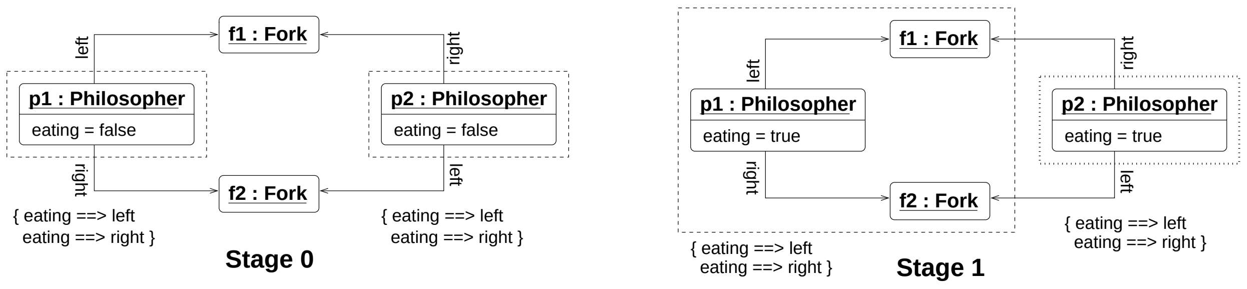

Figure 1. Realm dynamics

units in this stage. By contrast, in **Stage 1** of **Figure 1**, unit p_1 can access either or both of the units f_1 and f_2.

Each synchronization unit declares a synchronization constraint, which is used to infer, from the given unit's current state, the set of units to which it requires exclusive access. The constraint itself is a limited propositional formula over two types of variables, called *unit variables* and *condition variables*. Each unit variable names a potentially distinct dependency upon some supplier unit, i.e., the supplier unit to which the unit variable is currently bound. For example, p_1 declares two unit variables, `left` and `right`, whose bindings we depict graphically as named links that connect the client to the supplier. *Condition variables* encode synchronization-relevant states within a unit's lifetime. We depict these variables graphically as boolean-valued attributes, e.g., `eating`.

Formulas in curly braces ('{' and '}') depict synchronization constraints, which are interpreted as describing the unit's data-access needs in terms of its synchronization state. As a unit cycles through its various synchronization states, the set of units that it *entails*, i.e., those units which it may directly access from within the given state, changes. Consider, for example, p_1's constraint, which is read, "`eating` entails `left` and `eating` entails `right`." According to this constraint, p_1 may access f_1 (the unit referenced by its `left` variable) and f_2 (the unit referenced by its `right` variable) when the `eating` variable is true, and moreover it will not access either unit when this variable is false. Unit p_2 declares an analogous constraint.

Constraints relate to realms and the synchronization of processes as follows. We say that a realm is *sufficient* if it contains all supplier units that, according to the constraints of the units in the realm, the process may access; and that it is *minimally sufficient* if it contains only a root unit and units that are needed for it to be sufficient. For example, the two realms in **Stage 0** of **Figure 1** are minimally sufficient because they contain only a root unit (either p_1 or p_2) whose constraint states that no shared units are accessed while `eating` is false. By contrast, the realm that contains p_2 is not sufficient in **Stage 1** because p_2's constraint states that it may access both its left and right forks, but these units are not in the realm. Our model dictates that a process is *enabled* if its realm is minimally sufficient; otherwise, it is *blocked*. Graphically, we depict the realm of a blocked process using a dotted (as opposed to a dashed) outline.

Because sufficiency depends on the values of condition variables and unit variables, an operation that modifies these variables may affect the set of units that a process may access. We call such operations *realm affecting* because they trigger renegotiation of the affected constraints, thereby affecting the contents of realms. Such renegotiation involves the migration of units among realms. For example, the realm of the process running within p_1 is expanded in going from **Stage 0** to **Stage 1** as a result of modifying the value of `eating`. We say that f_1 and f_2 *migrate* into the realm containing p_1 to satisfy p_1's constraint.

To extend an OO language with Szumo requires adding syntax for declaring synchronization constraints and adding (or reusing) mechanisms for creating and configuring synchronization units and realms. Parsimony behooves us to add a minimal number of new features. Indeed some ordinary operations are made to trigger corresponding operations on units and realms, e.g., the construction of a new process object by a program triggers the construction of a new realm. However, to capitalize on these side-effecting operations requires a clean semantic model that allows the programmer to reason about what is triggered by each operation and about the consequences of those actions.

3 Foundations of our semantic model

The key to separately defining and then composing the *BaseView* and *SynchView* was to use a shared framework for structuring these specifications. This framework derives from the structure of a *metamodel* of object-oriented programs, i.e., a model of models of such programs. Our metamodel is defined as a four-tuple of Z schemas—one that represents type information, one that represents instance information, one that imposes a conformance invariant between types and instances, and a frame condition that imposes invariants over instance operations. We found this four-way

$$
\begin{array}{|l}
\hline \mathit{ClassModel}[\mathit{CLS}, \mathit{ASC}, \mathit{ATR}, \mathit{TYP}] \\
\mathit{Classes} : \mathbb{P}\,\mathit{CLS} \\
\mathit{attrType} : \mathit{ATR} \nrightarrow \mathit{TYP} \\
\mathit{attrOfClass} : \mathit{ATR} \nrightarrow \mathit{CLS} \\
\mathit{assoc} : \mathit{ASC} \nrightarrow (\mathit{CLS} \times \mathit{CLS}) \\
\hline \ldots \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \mathit{ObjModel}[\mathit{CLS}, \mathit{OBJ}, \mathit{ASC}, \mathit{LNK}, \mathit{ATR}, \mathit{VAL}] \\
\mathit{objClass} : \mathit{OBJ} \nrightarrow \mathit{CLS} \\
\mathit{objData} : \mathit{OBJ} \nrightarrow \mathit{ATR} \nrightarrow \mathit{VAL} \\
\mathit{linkAssoc} : \mathit{LNK} \nrightarrow \mathit{ASC} \\
\mathit{linkSource} : \mathit{LNK} \nrightarrow \mathit{OBJ} \\
\mathit{linkDest} : \mathit{LNK} \nrightarrow \mathit{OBJ} \\
\hline \ldots \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \mathit{ObjConformsToClass}[\mathit{CLASS}, \mathit{OBJ}, \mathit{ASSOC}, \mathit{LINK}, \mathit{ATTR}, \mathit{TYPE}, \mathit{VAL}] \\
\mathit{ClassModel}\ [\ \mathit{CLASS}, \mathit{ASSOC}, \mathit{ATTR}, \mathit{TYPE}\ ] \\
\mathit{ObjModel}\ [\ \mathit{CLASS}, \mathit{OBJ}, \mathit{ASSOC}, \mathit{LINK}, \mathit{ATTR}, \mathit{VAL}\ ] \\
\hline \operatorname{ran}(\mathit{objClass}) \subseteq \mathit{Classes}\ \wedge\ \operatorname{ran}(\mathit{linkAssoc}) \subseteq \operatorname{dom}(\mathit{assoc}) \\
\ldots \\
\hline
\end{array}
\qquad
\begin{array}{l}
\mathit{StableMap}[D, R] \mathrel{\widehat=} \\
\quad [\ \mathit{map} : D \nrightarrow R; \\
\qquad \mathit{map}' : D \nrightarrow R \mid \\
\qquad\quad \operatorname{dom}(\mathit{map} \cap \mathit{map}') = \\
\qquad\quad \operatorname{dom}(\mathit{map}) \cap \operatorname{dom}(\mathit{map}')\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{ObjModelFC}[\mathit{CLASS}, \mathit{OBJ}, \mathit{ASSOC}, \mathit{LINK}, \mathit{ATTR}, \mathit{VAL}] \mathrel{\widehat=} \\
\quad [\ \Delta\mathit{ObjModel}\ [\ \mathit{CLASS}, \mathit{OBJ}, \mathit{ASSOC}, \mathit{LINK}, \mathit{ATTR}, \mathit{VAL}\ ] \mid \\
\qquad \mathit{StableMap}\ [\ \mathit{OBJ}, \mathit{CLASS}\ ][\ \mathit{objClass}/\mathit{map}, \mathit{objClass}'/\mathit{map}'\ ]\ \wedge \\
\qquad \mathit{StableMap}\ [\ \mathit{LINK}, \mathit{ASSOC}\ ][\ \mathit{linkAssoc}/\mathit{map}, \mathit{linkAssoc}'/\mathit{map}'\ ]\ \wedge \\
\qquad \mathit{StableMap}\ [\ \mathit{LINK}, \mathit{OBJ}\ ][\ \mathit{linkSource}/\mathit{map}, \mathit{linkSource}'/\mathit{map}'\ ]\ ]
\end{array}
$$

Table 1. Generic class/object meta-model.

partitioning to be useful for defining individual views and for composing them. The remainder of this paper unfolds and illustrates the consequences of this key idea.

Recall how Z can be used to model a program $\mathcal{P}$ as an *abstract data type* of the form:

$$(P, PInit, \{i : I \bullet POp_i\})$$

Here, P is a schema that represents the set of conceivable program states, *PInit* is an *initialization schema, I* is a finite set of indices, and the indexed set contains schemas that represent operations over P. In our semantics, the set of conceivable states instantiates a metamodel of object-oriented programs, rather than a model of an individual program. That is, our equivalent to P represents *models* of classes and objects in an object-oriented program. Likewise, our equivalent to *PInit* represents initial configurations of programs whereby only a single object exists, and the operations in the indexed set represent *metamodel operations,* which we now describe in more detail.

Metamodel Operations model the construction/deletion and configuration of objects and data values. We draw a sharp distinction between a metamodel operation and a method that one normally associates with classes in an object-oriented program, e.g., a method that retrieves the head of a list. A metamodel operation may represent one step in the execution of a method on some object or multiple concurrent steps of different methods. For example, one metamodel operation could specify the construction of an object by one process concurrent with the update of an attribute of some other object by a different process. This abstract treatment of program operations is meaningful because our purpose in modeling is to understand the composition of language features rather than to understand the meaning of a particular program in a Szumo-extended language.

Table 1 depicts the structure of our metamodel of programs. Structurally, this model is reminiscent of prior work in formalizing UML in Z [14] and in Object-Z [18], except that we provide a frame condition to constrain the behaviors of metamodel operations and our schemas are generic so as to be easily reused among multiple views. The generic schema *ClassModel* models the static typing information of programs in a class-based language as if this information had been specified in a class diagram using the conventions of [24].[5] The set *Classes* records all of the classes that a programmer has declared. The partial function *attrType* maps an attribute (element of the generic parameter set *ATR*) to its declared type (element of the generic parameter set *TYP*), and the function *attrOfClass* maps an attribute to the class (element of the generic parameter set *CLS*) in which that at-

[5]Most notably: Class attributes are of only primitive types; references to objects are modeled using links of associations.

tribute is declared. Finally, the function *assoc* models the set of associations as a mapping from an association (element of the generic parameter *ASC*) to a pair of associated classes.

The generic *ObjModel* schema defines configurations of objects, each of which may be attributed with values, and links, each of which connects two objects. This schema is parameterized by several of the same sets that parameterized *ClassModel*, and in addition: *OBJ*, *LNK*, and *VAL*, which represent objects, links, and attribute values respectively. Here, *objClass* associates each object with the class it instantiates, and *objData* associates each object with a binding of attributes to values. The partial functions *linkAssoc*, *linkSource*, and *linkDest* record analogous information about links. This model captures the notion that a program is a dynamic configuration of attributed objects.

The generic schema *ObjConformsToClass* codifies what we expect to be true of an object model that conforms to a given class model. It states that every object and link in the object model instantiates some class or association in the class model, and that attribute values (resp. link source and destination objects) are of the appropriate type (resp. class) according to the class model. Here and elsewhere in our semantics, we express type–instance conformance using a separate invariant because doing so simplifies the specification of both the type and instance state schemata (e.g., *ClassModel* and *ObjModel*). This separation is particularly useful for integrating declarative concurrency extensions, such as Szumo, because the syntactic extensions adorn the definition of classes (i.e., types) but have an implicit effect on objects (i.e., instances).

Finally, we define frame conditions that impose soundness constraints on the invocation of metamodel operations. The generic schema *ObjModelFC*[6] prohibits program operations from changing the class to which an object belongs, the association to which a link belongs, or the source object of a link. We refer to this condition as stating that the *objClass*, *linkAssoc*, and *linkSource* mappings are *stable* under change. The stability property applies to other mappings in other views; thus we abstract it here using a generic schema definition.

To summarize, our metamodel comprises four distinct kinds of schemas:

1. a space of conceivable *types* and relations among types (e.g., *ClassModel*),

2. a space of conceivable *instances* and configurations of instances (e.g., *ObjModel*),

3. a *conformance invariant* that relates the space of conceivable instances to the space of conceivable types (e.g., *ObjConformsToClass*), and

[6]Here and in the sequel, schemas whose names end in "FC" denote frame conditions.

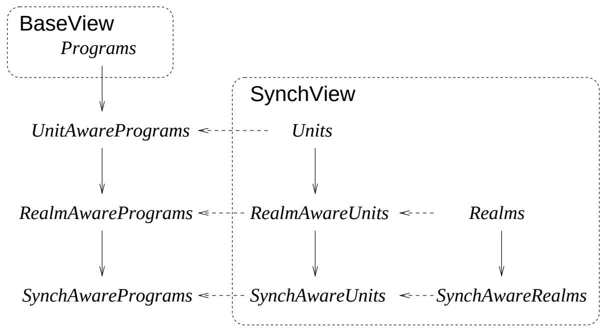

Figure 2. View-based semantics of synchronization-aware programs

4. an *instances frame condition* that must be conjoined with any metamodel operation (e.g., *ObjModelFC*).

Because we are interested only in extensions to statically-typed languages, there are no operations over the *types* component of our model and thus no need to specify "types frame conditions" other than that the types are unaffected by an operation. Organizing the specifications of both the *BaseView* and the *SynchView* according to these conventions greatly simplifies their composition.

4. Architecture of our semantic model

Figure 2 depicts the high-level architecture of our view-based semantics of synchronization-aware programs (i.e., the composition of *BaseView* and *SynchView*). Each italicized entity indicates a cohesive view, which is organized according to the structuring conventions described in **Section 3**. Notice that *SynchView* is itself composed from several smaller views and that this decomposition of *SynchView* is used to incrementally compose it with the *BaseView*. We now briefly describe the *BaseView* and use it to illustrate one level of view composition, namely the composition of base programs and unit programs into *UnitAwarePrograms*. Using this example, we then discuss some of the interesting properties of the architecture.

4.1. The base view

Our *BaseView* comprises a specification called *Programs*, which defines metamodels of object-oriented programs that are concurrent but synchronization unaware. Commensurate with our conventions for modularizing views as schema tuples, we first define the language's static type information. The declarations:

$$[PCLASS, PASSOC, PATTR]$$

$$
\begin{array}{ll}
\mathit{PTYPE} & ::= \mathsf{pBool} \\
\mathit{PCond} & ::= \ldots \\
\mathit{PLocalCond} & ::= \ldots
\end{array}
$$

$$
\begin{array}{rl}
\mathit{PConstr} ::= & \mathsf{pcTrue} \mid \mathsf{pcRef}\langle\!\langle \mathit{PASSOC} \rangle\!\rangle \\
\mid & \mathsf{pcRefWhen}\langle\!\langle \mathit{PASSOC} \times \mathit{PCond} \rangle\!\rangle \\
\mid & \mathsf{pcENTAILS}\langle\!\langle \mathit{PLocalCond} \times \mathit{PConstr} \rangle\!\rangle \\
\mid & \mathsf{pcAND}\langle\!\langle \mathit{PConstr} \times \mathit{PConstr} \rangle\!\rangle
\end{array}
$$

introduce basic types needed to instantiate the various metamodel schemas and a free type that models declarations in our language of synchronization constraints. Schema *ProgTypes* then models the static type system as follows:

$$
\begin{array}{|l}
\hline \mathit{ProgTypes} \\
\mathit{ClassModel}\ [\ \mathit{PCLASS}, \mathit{PASSOC}, \mathit{PATTR}, \mathit{PTYPE}\ ] \\
\mathit{procClass} : \mathit{PCLASS} \\
\mathit{SynchClasses} : \mathbb{P}\,\mathit{PCLASS} \\
\mathit{RealmClasses} : \mathbb{P}\,\mathit{PCLASS} \\
\mathit{synchClassConst} : \mathit{PCLASS} \nrightarrow \mathit{PConstr} \\
\hline
\mathit{SynchClasses} \subseteq \mathit{Classes} \\
\mathit{RealmClasses} \subseteq \mathit{SynchClasses} \\
\mathit{procClass} \in \mathit{Classes} \setminus \mathit{SynchClasses} \\
\mathrm{dom}(\mathit{synchClassConst}) = \mathit{SynchClasses} \\
\hline
\end{array}
$$

In addition to instantiating the generic schema *ClassModel*, *ProgTypes* introduces four schema variables, *procClass*, *SynchClasses*, *RealmClasses*, and *synchClassConst*. The distinguished *procClass* denotes a class which, when instantiated, should produce both a new object and a new process that is associated with that object. The sets *SynchClasses* and *RealmClasses* contain programmer-designated *synchronization classes* and *realm classes* respectively. When instantiated, a class in *SynchClasses* should produce both a new object and a new synchronization unit that contains the new object. In the sequel we will refer to the object whose lifetime coincides with the lifetime of the corresponding unit as the *root object* of the unit. A realm class is a synchronization class whose instantiated units may serve as the *root unit* of a realm. The root object of a unit is a distinct concept from the root unit of a realm. Every instance of a synchronization class is a root object of some unit; this unit will be the root unit of a realm precisely when the synchronization class is also a realm class. Each synchronization class is associated with a concurrency constraint (i.e., element of *PConstr*).

Notice that while *ProgTypes* defines mechanisms for representing Szumo declarations, none of the semantics of these declarations are depicted in this schema. This is by design: To properly specify their effects requires an understanding of units and realms, which are not part of the *BaseView*.

Next, we model the space of instance configurations in our Szumo-extended language. The declarations:

$$[\mathit{POBJ}, \mathit{PLINK}, \mathit{PROCESS}]$$

$$\mathit{PVAL} ::= \mathsf{pFalse} \mid \mathsf{pTrue}$$

introduce a space of objects, links, processes, and attribute values, which for brevity in this paper we restrict to booleans. Schema *ProgInstances* models the dynamic configuration of objects and processes in programs written in an object-oriented language:

$$
\begin{array}{|l}
\hline \mathit{ProgInstances} \\
\mathit{ObjModel}\ [\ \mathit{PCLASS}, \mathit{POBJ}, \mathit{PASSOC}, \\
\qquad \mathit{PLINK}, \mathit{PATTR}, \mathit{PVAL}\ ] \\
\mathit{procToObj} : \mathit{PROCESS} \rightarrowtail\!\!\!\!\!\!+\;\; \mathit{POBJ} \\
\hline
\mathrm{ran}(\mathit{procToObj}) \subseteq \mathrm{dom}(\mathit{objClass}) \\
\hline
\end{array}
$$

In addition to instantiating the generic metamodel schema *ObjModel*, *ProgInstances* introduces an injective mapping (*procToObj*) between processes and objects. The objects in the range of this mapping must be instances of class *procClass*; and indeed every instance of class *procClass* must be represented in this mapping. This type–instance conformance invariant is formalized as follows:

$$
\begin{array}{|l}
\hline \mathit{ProgConforms} \\
\mathit{ProgTypes} \\
\mathit{ProgInstances} \\
\hline
\mathit{ObjConformsToClass}\ [\ \mathit{PCLASS}, \mathit{POBJ}, \\
\qquad \mathit{PASSOC}, \mathit{PLINK}, \\
\qquad \mathit{PATTR}, \mathit{PTYPE}, \mathit{PVAL}\ ] \\
\mathrm{ran}(\mathit{procToObj}) = \\
\qquad \mathrm{dom}(\mathit{objClass} \rhd \{\mathit{procClass}\}) \\
\hline
\end{array}
$$

Finally, the schema *ProgInstancesFC* incorporates the generic frame condition over object models and further requires that the *procToObj* mapping be stable, which means that the binding of a process to an object is invariant modulo the creation/deletion of processes.

$$
\begin{array}{l}
\mathit{ProgInstancesFC} \mathrel{\widehat{=}} \\
\quad \mathit{ObjModelFC}\ [\ \mathit{PCLASS}, \mathit{POBJ}, \mathit{PASSOC}, \\
\qquad\qquad \mathit{PLINK}, \mathit{PATTR}, \mathit{PVAL}\ ] \ \wedge \\
\quad \mathit{StableMap}\ [\ \mathit{PROCESS}, \mathit{POBJ}\ ] \\
\qquad\qquad [\ \mathit{procToObj}/\mathit{map}, \mathit{procToObj}'/\mathit{map}'\ ]
\end{array}
$$

The *Programs* view may now be expressed formally as:

$$
\begin{array}{l}
\mathit{Programs} == (\ \mathit{ProgTypes}, \mathit{ProgInstances}, \\
\qquad\qquad \mathit{ProgConforms}, \mathit{ProgInstancesFC}\ )
\end{array}
$$

$$[\mathit{UCLASS}, \mathit{UOBJ}, \mathit{UASSOC}, \mathit{ULINK}, \mathit{UATTR}]$$

$$\begin{array}{lll}
\mathit{UTYPE} & ::= & \mathsf{uBool} \\
\mathit{UVAL} & ::= & \mathsf{uFalse} \mid \mathsf{uTrue} \\
\mathit{UConstr} & ::= & \mathsf{ucTrue} \\
 & \mid & \mathsf{ucRef}\langle\!\langle \mathit{UASSOC} \rangle\!\rangle \\
 & \mid & \mathsf{ucRefWhen}\langle\!\langle \mathit{UASSOC} \times \mathit{UCond} \rangle\!\rangle \\
 & \mid & \ldots
\end{array}$$

$$\begin{array}{|l}
\textit{UnitTypes} \\
\hline
\mathit{ClassModel}\ [\ \mathit{UCLASS}, \mathit{UASSOC}, \\
\qquad\qquad \mathit{UATTR}, \mathit{UTYPE}\ ] \\
\qquad\qquad\qquad [\ \mathit{UnitClasses}/\mathit{Classes}, \\
\qquad\qquad\qquad\ \ \mathit{uattrOfUClass}/\mathit{attrOfClass}, \\
\qquad\qquad\qquad\ \ \mathit{uattrUType}/\mathit{attrType}, \\
\qquad\qquad\qquad\ \ \mathit{uassoc}/\mathit{assoc}\ ] \\
\mathit{uClassConstraint} : \mathit{UCLASS} \pfun \mathit{UConstr} \\
\hline
\operatorname{dom}(\mathit{uClassConstraint}) = \mathit{UnitClasses} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{UnitInstances} \\
\hline
\mathit{ObjModel}[\ \mathit{UCLASS}, \mathit{UOBJ}, \mathit{UASSOC}, \\
\qquad\qquad \mathit{ULINK}, \mathit{UATTR}, \mathit{UVAL}\ ] \\
\qquad\qquad\quad [\ \mathit{unitClass}/\mathit{objClass}, \\
\qquad\qquad\quad\ \ \mathit{unitData}/\mathit{objData}, \\
\qquad\qquad\quad\ \ \mathit{unitLinkAssoc}/\mathit{linkAssoc}, \\
\qquad\qquad\quad\ \ \mathit{unitLinkSource}/\mathit{linkSource}, \\
\qquad\qquad\quad\ \ \mathit{unitLinkDest}/\mathit{linkDest}\ ] \\
\mathit{unitConstraint} : \mathit{UOBJ} \pfun \mathit{UConstr} \\
\hline
\operatorname{dom}(\mathit{unitConstraint}) = \operatorname{dom}(\mathit{unitClass}) \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{UnitInstancesFC} \mathrel{\widehat{=}} \\
\quad \mathit{ObjModelFC}\ [\ \mathit{UCLASS}, \mathit{UOBJ}, \ldots\ ][\ldots]\ \ \land \\
\quad \mathit{StableMap}\ [\ \mathit{UOBJ}, \mathit{UConstr}\ ] \\
\qquad\qquad\qquad [\ \mathit{unitConstraint}/\mathit{map}, \\
\qquad\qquad\qquad\ \ \mathit{unitConstraint}'/\mathit{map}'\ ]
\end{array}$$

$$\begin{array}{|l}
\textit{UnitConforms} \\
\hline
\mathit{UnitTypes} \\
\mathit{UnitInstances} \\
\hline
\mathit{ObjConformsToClass}[\ \mathit{UCLASS}, \mathit{UOBJ}, \ldots][\ldots] \\
\forall\, u : \operatorname{dom}(\mathit{unitConstraint}) \bullet \\
\quad \mathit{unitConstraint}(u) = \\
\qquad \mathit{uClassConstraint}(\mathit{unitClass}(u)) \\
\hline
\end{array}$$

Table 2. Components of the *Units* view.

$$\begin{array}{|l}
\mathit{Clusters}[\mathit{ELEM}, \mathit{CLUSTER}] \\
\hline
\mathit{root} : \mathit{CLUSTER} \pinj \mathit{ELEM} \\
\mathit{in} : \mathit{ELEM} \pfun \mathit{CLUSTER} \\
\hline
\mathit{root}^{\sim} \subseteq \mathit{in} \\
\operatorname{dom}(\mathit{root}) = \operatorname{ran}(\mathit{in}) \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{StrictClustersFC}[E, C] \mathrel{\widehat{=}} \\
\quad \mathit{StableMap}[C, E][\mathit{root}/\mathit{map}, \mathit{root}'/\mathit{map}']\ \ \land \\
\quad \mathit{StableMap}[E, C][\mathit{in}/\mathit{map}, \mathit{in}'/\mathit{map}']
\end{array}$$

Table 3. Anciliary definitions

4.2 Unit programs

The *Units* view models synchronization units in the abstract—i.e., as object-like structures irrespective of how they relate to program objects. We define it as:

$$\begin{array}{l}
\mathit{Units} == (\ \mathit{UnitTypes}, \mathit{UnitInstances}, \\
\qquad\qquad\quad \mathit{UnitConforms}, \mathit{UnitInstancesFC}\ )
\end{array}$$

in a manner analogous to the definition of *Programs*. *Units* is essentially just an instantiation of the metamodel with a few additional syntactic extensions. **Table 2** depicts the declarations and constituent schemas.

Units are "object-like" structures; thus the basic types introduce unit classes and instances (*UCLASS* and *UOBJ*), unit associations and links (*UASSOC* and *ULINK*) and unit attributes (*UATTR*). The free type *UConstr* defines the language of synchronization constraints, phrased in the terminology of unit classes rather than program classes. The mapping *uClassConstraint* in schema *UnitTypes* associates these constraints with unit classes, and the mapping *unitConstraint* in schema *UnitInstances* associates them with individual units.[7] The conformance invariant relates these mappings in the obvious way.

4.3 View composition: Unit-aware programs

We now show how to incorporate one level of view composition. A composite view is specified by composing two or more other views in our framework. **Figure 2** depicts five composite views, one of which defines the *unit-aware program* abstraction. A unit-aware program is a base program whose objects cluster into synchronization units. In terms of behavior, it can be thought of as a base program running alongside a unit program with operations on the

[7]This latter mapping, while not strictly necessary, simplifi es the composition of *Units* with *Realms* (details not provided in this paper).

former implicitly affecting the latter. The structure and behavior of unit-aware programs is defined by a view called *UnitAwarePrograms*, which is formed by composing the *Programs* and *Units* views. We now step through this composition to give a sense of how it works and how composition is simplified by our structuring conventions. Space limitations prevent our showing the integration of realm- and synchronization- awareness, but these are incorporated in a similar manner.

The "types" component of the unit-aware programs view is defined as follows:

$$\begin{array}{|l}
\textit{UnitAwareProgTypes} \\
\hline
\textit{ProgTypes} \\
\textit{UnitTypes} \\
\textit{synchClassMap} : \textit{PCLASS} \rightarrowtail\!\!\!\!\twoheadrightarrow \textit{UCLASS} \\
\hline
\mathrm{dom}(\textit{synchClassMap}) = \textit{SynchClasses} \\
\ldots \\
\hline
\end{array}$$

Notice that we conjoin the analogous schemas from the *Programs* and *Units* views and relate them using an invariant. Here, the mapping *synchClassMap* associates classes that were designated as synchronization classes in the *BaseView* with unit classes in the *SynchView*. Other invariants (elided for brevity) impose a correspondence between unit classes and associations and program classes and associations.

The instances component of this view is defined by conjoining the analogous "instances" components from the *Programs* and *Units* views and relating these instances by requiring units to cluster program objects. A *cluster* is essentially a set of entities with one entity distinguished as a representative (called *root*) for the others (**Table 3**). Each unit is a cluster of program objects, one of which is designated as the root of the unit:

$$\begin{array}{|l}
\textit{UnitAwareProgInstances} \\
\hline
\textit{ProgInstances} \\
\textit{UnitInstances} \\
\textit{Clusters}[\textit{POBJ}, \textit{UOBJ}][\textit{unitRoot}/\textit{root}, \\
\qquad\qquad\qquad\qquad \textit{objInUnit}/\textit{in}] \\
\hline
\mathrm{dom}(\textit{objData}) = \mathrm{dom}(\textit{objInUnit}) \\
\mathrm{dom}(\textit{unitRoot}) = \mathrm{dom}(\textit{unitData}) \\
\hline
\end{array}$$

This schema imposes two invariants: Every object must be an instance of some unit and the units referenced in the clustering mappings must coincide with those referenced in the *Units* view. Consequently, a program object cannot exist without being clustered into some unit, and every unit must cluster some non-empty set of program objects.

The conformance invariant (*UnitAwareProgConforms*) connects the set of units to the set of objects that instantiate a class in *SynchClasses*.

$$\begin{array}{|l}
\textit{UnitAwareProgConforms} \\
\hline
\textit{UnitAwareProgTypes} \\
\textit{UnitAwareProgInstances} \\
\textit{ProgConforms} \\
\textit{UnitConforms} \\
\hline
\mathrm{dom}(\textit{objClass} \rhd \textit{SynchClasses}) = \mathrm{ran}(\textit{unitRoot}) \\
\hline
\end{array}$$

Finally, the frame condition imposes a "strictness" property on the clusters, meaning that the "root status" of an object in a unit may not vary over the lifetime of the unit and that objects cannot migrate among units.

$$\begin{array}{l}
\textit{UnitAwareProgInstancesFC} \mathrel{\widehat=} \\
\quad [\Delta \textit{UnitAwareProgInstances} \mid \\
\qquad \textit{ProgInstancesFC} \land \textit{UnitInstancesFC} \land \\
\qquad \textit{StrictClustersFC}[\textit{POBJ}, \textit{UOBJ}] \\
\qquad\qquad\qquad [\, \textit{unitRoot}/\textit{root}, \textit{unitRoot}'/\textit{root}', \\
\qquad\qquad\qquad\ \ \textit{objInUnit}/\textit{in}, \textit{objInUnit}'/\textit{in}']]
\end{array}$$

4.4 Properties of the architecture

While only a small part of the larger composition of views, the specification of *UnitAwarePrograms* serves to illustrate several useful properties of our architecture. Each component of a composite view is built up systematically from the corresponding components of the more primitive views that are being composed. Moreover, what is added beyond the inclusion of dependent components is not large (e.g., the addition of the *synchClassMap* in the types component, the addition of the *Clusters* invariant in the instances component, one line in the predicate part of the conformance-invariant component, and one additional conjunct in the frame-condition component). These properties are typical of the other composite specifications named in **Figure 2**.

By structuring each view according to the our conventions, we were able to decompose the *SynchView* into a two-dimensional complex of views that nicely separates concerns and simplifies composition with the *BaseView*. For example, programs in this *BaseView* are not *synchronization-aware*, despite having been adorned with the declarative features of Szumo. These programs are made synchronization-aware by incrementally incorporating the components of the *Programs* view with corresponding components from the *Units*, *RealmAwareUnits*, and *SynchAwareUnits* views, as indicated by the arrows in the first column of the diagram.

By design, each named view contributes some significant capability to the overall semantics without mixing in unnecessary details. As indicated by the diagram, the *SynchView* is itself a complex of specifications, and indeed we found it quite difficult to compose our early specifications of this view with the *BaseView*. However, by carefully decomposing the *SynchView* specifications as depicted in the diagram, we found that we could exploit this structure to incrementally compose it with the *BaseView* specifications.

The rows and columns of this structure constitute semantically meaningful but mutually cross-cutting concerns. The columns define the various conceptual models that a programmer must manage when building systems in our extended language. For example, the second column defines the conceptual model of *unit programs*, which are more abstract (and compact and easier to visualize) than the corresponding base programs. This column begins with the most basic model of a unit program—one whose operations can modify the state of a collection of units, with no notion of realms or synchronization. Subsequent refinements incrementally incorporate these additional concepts. For example, *RealmAwareUnits* uses a relation for judging the satisfiability of synchronization constraints to partition units into disjoint realms. The refinement of this specification into *SynchAwareUnits* ties the enabling of realms to their minimal sufficiency.

By contrast, the rows encapsulate and define cross-cutting features of our extended language. For example, the third row defines *realm-awareness*, starting with a most basic model of realm programs (i.e., *Realms*). Concepts in this most basic view are incrementally integrated into the other conceptual models (e.g., *Units* and *Programs*) that a programmer must manage when building applications in our extended language.

5. Discussion

Formal semantics of real programming languages are large and complicated, and some of the details that such a semantics would prescribe are irrelevant to the integration of our concurrency model. We therefore opted to use a very abstract *BaseView*—one that attempts to capture only the "essence" of the object-oriented languages we wish to extend.[8] This paper contributes a view-based framework for incrementally specifying complex language extensions in a form that demonstrates clean separation of concerns.

Early work on specification in Z using views [6, 1, 10] treats issues of composition (called *amalgamation* in these papers) of unrestricted views and consistency between them. In contrast, we impose additional structure on component views, which simplifies composition and more clearly delineates the specification of the new language features and the specification of how they affect and are affected by existing features of a language.

The resulting view-based semantics of synchronization-aware programs is modular and extensible. Modular semantics are of interest to language designers. For example, Reppy extends the formal semantics for a subset of Standard ML using combinators [23]. Mosses and Musicante extend a sequential language with message-based concurrency primitives, and then show that an action semantics of the sequential language can be augmented without modification to produce a definition of the extended language [21].

Domain-specific languages frequently extend an existing base language. Ekman and Hedin [12, 13] describe a tool for creating reusable modules for domain-specific language extensions from algebraic specifications. Hudak [15] uses monads for a similar purpose. We do not propose to generate tools from our view-based semantics.

It is well known that separating program aspects can introduce errors if they are not sufficiently orthogonal because of the difficulty of reasoning about unanticipated interactions [11]. Schmied and Hauck propose extending aspects with meta-level pre- and post-conditions resembling Z schemas in order to be able to cleanly compose them [25]. Brichau et al. [7] use logic metaprogramming to create composable domain-specific aspect languages.

The organization of our specification (**Figure 2**) resembles those proposed by advocates of *multi-dimensional separation of concerns* (MDSOC) [22]. MDSOC organizes the space of basic software elements by grouping them along different dimensions, such as by classes or by features. Batory et al. [2] also note that multi-dimensional models produce more compact program specifications, which can be composed in very powerful ways. Our approach applies the same principle to a specification of a language extension instead of to executable program code, with the rows and columns of our design each supporting a different dimension.

Our experience in specifying the semantics of negotiation affirms the sentiments of Bjorner and Jones, who argue that formal models should be constructed *during* the definition of a language rather than after the fact [5]. In fact, as a result of the work presented in this paper, we made several modifications to our synchronization constraint language. Most notably, we abandoned the use of disjunctions because of problems in providing a clean semantics—despite the fact that we had already implemented support for disjunctions in early prototypes of our Eiffel extension. We were fortunate to discover these problems before releasing the language and its compiler to a large audience.

Acknowledgements Partial support for this research provided by NSF grants CCR-9984727 and EIA-0000433 and by the Office of Naval Research grant N00014-01-0744.

[8] Of course, over-abstraction may mask a critical but unforeseen feature interaction. For example, our abstraction does not allow us to reason about the potential for inheritance anomalies. Our purpose in this paper is to use the formal model to understand how one should "think about" the integration of object-oriented and Szumo concepts, and for this purpose, abstraction is critical.

References

[1] M. Ainsworth, A. H. Cruickshank, P. J. L. Wallis, and L. J. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, 1994.

[2] D. Batory, J. Liu, and J. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proc. of ESEC/FSE*, pages 48–57. ACM Press, 2003.

[3] 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.

[4] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, 2000.

[5] D. Bjorner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[6] H. Bowman and J. Derrick. Modelling distributed systems using Z. In K. M. George, editor, *ACM Symposium on Applied Computing*, pages 147–151. ACM Press, 1995.

[7] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 110–127. Springer-Verlag, 2002.

[8] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proc. of the Workshop on Foundations of Aspect-Oriented Languages (FOAL'02)*, 2002.

[9] R. DeLine and M. Fahndrich. Typestates for objects. In *Proc. 18th ECOOP*, 2004.

[10] J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 413–424. Chapman & Hall, Feb. 1995.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of International conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.

[12] T. Ekman. Separation of concerns in compiler construction using JastAdd II. In *Proc. of AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2004.

[13] T. Ekman and G. Hedin. Reusable language specification modules in JastAdd II. In *Workshop on Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004.

[14] R. France, A. Evans, and K. Lano. The UML as a formal modeling notation. In *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81, 1997.

[15] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134. IEEE Computer Society, 1998.

[16] D. Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Meth.*, 4(4):365–389, 1995.

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

[18] S.-K. Kim and D. Carrington. A formal mapping between UML models and Object-Z specifications. In *Proc. of ZB2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 2–21, 2000.

[19] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[20] C. Lopes. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.

[21] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[22] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Comm. ACM*, 44(10):43–50, 2001.

[23] J. H. Reppy. *Concurrent Programmin in ML*. Cambridge University Press, 1999.

[24] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[25] A. I. Schmied and F. J. Hauck. Composing non-orthogonal aspects. In *The 13th Workshop for PhD Students in Object-Oriented Systems*. Springer-Verlag, 2003.

[26] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1992.

[27] 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.

[28] R. E. K. Stirewalt, L. K. Dillon, and R. Behrends. Semantics of the synchronization units model: Version 2.0. Technical Report MSU-CSE-06-22, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2006.

[29] P. Zave and M. Jackson. Conjunction as composition. *ACM Trans. Softw. Eng. Meth.*, 2(4):371–411, 1993.