

Towards a Modeling Environment for Composing Domain-Specific Modeling Languages: A Case Study on Controlling Traffic Lights

Joseph Porter

Institute for Software Integrated
Systems
Vanderbilt University
Nashville, Tennessee, USA 37203
jporter@isis.vanderbilt.edu

Janos Sztipanovits

Institute for Software Integrated
Systems
Vanderbilt University
Nashville, Tennessee, USA 37203
sztipaj@isis.vanderbilt.edu

Abstract. Domain-Specific Modeling Languages (DSMLs) play a fundamental role in the model-based design of embedded software and systems. While abstract syntax metamodeling enables the rapid and inexpensive development of DSMLs, the specification of DSML semantics is still a hard problem, particularly for models with heterogeneous models of computation. In this paper we consider issues associated with creating modeling tools which allow for the composition of semantic units. Key concepts are shown through a very simple case study involving traffic controllers.

1.0 Introduction

Semantics for domain-specific modeling languages (DSMLs) can be specified in a number of ways according to the choices of the language designer, the concepts in the application domain, and the requirements of a particular application area. Semantics can also be used for various purposes in design and analysis. This suggests the need for flexibility in describing semantics, both in detail and in the types of relationships that can be represented. Traditionally, programming language semantics are grouped into at least three categories in the literature:

1. Axiomatic semantics, which assign logical predicates to language constructs. Properties of the language or of a given program are then determined by assembling logical statements.
2. Denotational semantics, which associate specific mathematical structures with language constructs. Sometimes axiomatic semantics are grouped with denotational semantics, and many denotational frameworks include logical constructs (as do the programming languages themselves), or are built directly on logic.
3. Operational semantics, which describe programs using behavioral descriptions of language elements.

Modern language semantics practice includes more categories and sub-categories [8,9], which will not be considered here. For our purposes semantics described by

abstract machines are included in the operational category, though the categorization is only important conceptually. We emphasize that these semantic categories do not compete, rather we see them as different views into mathematical models describing the meaning of a DSML. Our particular focus will be on behavioral semantics, which are most naturally specified operationally.

Abstract state machines have been successfully used to specify semantics for DSMLs [3,4,5,6]. The Abstract State Machine Language (AsmL) from the Foundations of Software Engineering group at Microsoft Research [1] is an ideal semantics specification language for a number of reasons:

1. AsmL (as all ASM languages) is built on a carefully-constructed formal foundation. An operational description of semantics in ASML implies the existence of mathematical models for both the DSML itself and for the behavior of models created in the DSML. These implicit mathematical structures (or an appropriate subset) may be made explicit by defining mappings and developing model transformations into direct logical or other formal representations. This idea has important consequences for creating DSMLs for which system properties can be verified.
2. AsmL has rich data structures. This advantage is compatible with the idea that we seek to capture mathematical concepts clearly and concisely.
3. AsmL borrows syntax from engineering languages [2], so semantic specifications in ASML can be both brief and clear to those versed in common software development languages.
4. AsmL (and again all ASM languages) has a rich notion of concurrency. The ASM semantics dictate that all updates in a given step execute in parallel. This offers the least possible restriction for specifying semantics of languages with concurrency concepts (like synchrony) as well as for modeling and simulating distributed platforms.

DSML semantics are currently defined in terms of well-studied models of computation whose implementations are called *semantic units*. Semantic units express behaviors using modular sub-languages, such as finite state machines, timed automata, or various kinds of dataflow. Semantic anchoring is the process of associating concepts and constructs in a DSML with elements of one or more semantic units [3]. As an individual modeling language grows in size and complexity, the anchoring process becomes more involved and difficult to manage. This problem arises as numerous model concepts must be associated with entities in the semantic units. Existing semantic anchoring tools require this to be done by hand between the DSML metamodels and the semantic unit data models. An ideal approach should capture the equivalences using simple model composition operations and then to automate the generation of the remaining details.

In order to create automation tools for the semantic anchoring process, composition of DSML semantics is perhaps the largest issue. High-complexity modeling languages are built compositionally, with multiple levels of hierarchy and heterogeneous models of computation at different levels. As the control flow moves through a model, different components interact according to their specified behavioral models. We will discuss some concepts from other compositional models as well as providing simple examples to illustrate how composition can be specified.

2.0 Composition Concepts

We give a few samples of relevant research in modeling and behavioral composition:

- The Ptolemy II modeling framework is an experimental platform that implements heterogeneous composition of models of computation [12]. Models are created hierarchically, and a different model of computation can be chosen for each level of the hierarchy. Individual computational blocks (actors) have a well-defined and consistent execution interface which is invoked according to the chosen director block. The director implements the model of computation for the individual actors, interactions between actors at each level of the hierarchy, and interactions across the hierarchy. The directors include various data flow as well as automata models of computation.
- Interaction models [7] offer a more abstract framework from which we can borrow ideas. Each component interacts through ports, each of which offers a finite set of actions. Connections between components are analyzed in terms of the power sets of all possible actions on a particular connector. The valid interactions are subsets of the power sets, with validity determined by properties of the connection endpoints. Interaction models also separate individual component behavior from the handling of component interactions. Properties such as safety can be determined from interaction models.
- Partial order models [10,11,13] provide structures for representing concurrent behavior including sequential dependencies, independence, and conflicts between events which trigger or are triggered by a computing system. These denotational frameworks attempt to address composition issues in system models that include concurrent behavior and define equivalence with less abstract, intensional frameworks (e.g. state models and transition system).

For performing semantic composition in our modeling environment, we wish to use concepts from all of these approaches. We also give the following requirements:

1. We must be able to formally and flexibly associate concepts in a chosen model of computation with concepts in our DSML.
2. We must be able to cleanly separate individual component behavioral semantics from component interaction semantics in component-based DSMLs. Model segments with different models of computation should interact through a consistent and well-defined interface, and the specified interactions should be clear and easy to understand.
3. We must maintain the links to underlying formal (e.g. denotational and/or axiomatic) semantic models, despite all of the heterogeneity of behavior that may appear in our models. This will enable and even facilitate analysis and behavioral verification of critical system properties.
4. We must automate as much of the anchoring process as possible to keep the DSML designers from drowning in the details of large language compositions while retaining openness to changes.

3.0 Streets, Intersections, and Semantics

Consider the following metamodels for a DSML that describes traffic environments and controllers. The village metamodel (Fig. 1) contains streets and intersections, which have simple concurrent dataflow semantics. Streets are serial one-way buffers, so two street objects are used to describe a single physical street connecting a pair of intersections. An intersection has up to eight ports – one input and one output port for each of the four cardinal directions. This experimental language is used for experiments in concurrency modeling, so the intersection semantics require all enabled input ports in the intersection to concurrently transfer one waiting vehicle to its specified destination port. This corresponds roughly to real street intersections without traffic lights, with all of the attendant conflicts of blind drivers.

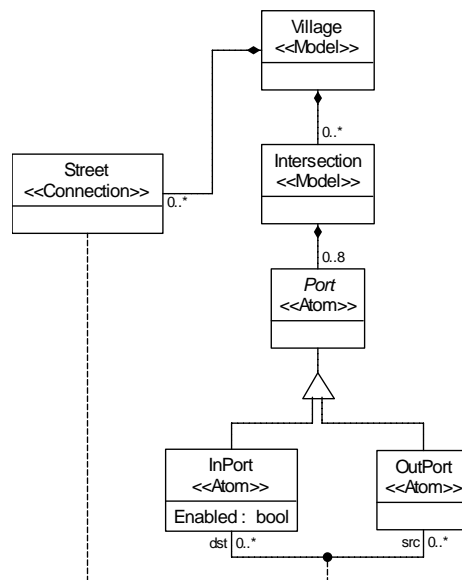


Fig. 1. Village metamodel with streets, intersections, and ports.

The vehicle metamodel (Fig. 2) describes paths of vehicles through the ports of the village. All vehicles travel independently - their aggregate semantics is that of independent concurrent automata. In this version of our experimental language, vehicles have only a single path – their controlling automata can not branch. This is not a requirement, rather a simplification allowing easy prediction of group behaviors.

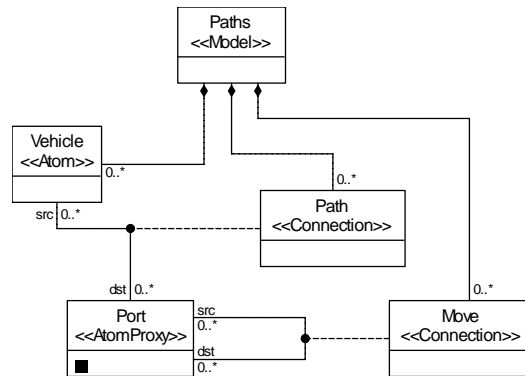


Fig. 2. Path metamodel for vehicles.

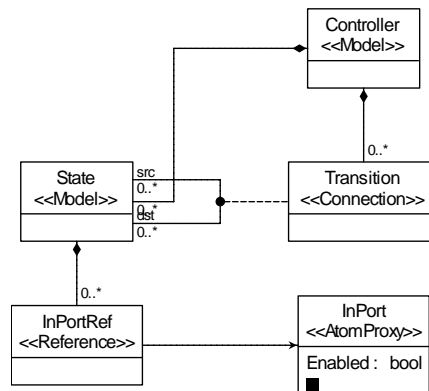


Fig. 3. Controller metamodel. States enable the input ports of intersections.

Fig. 3 contains the controller metamodel. This model is nothing more than a finite state automaton which enables the input ports of a particular intersection. A controller, forces the intersection behavior to only allow progress from input ports that are enabled by the current controller state. The function of the controller is to prevent conflicting updates in the intersection. All controllers execute independently, leading again to parallel finite automata semantics in the aggregate.

Two kinds of conflicts exist as vehicles cross intersections. A correct controller can only prevent conflicts arising from two vehicles crossing at the same time. The other type of conflict happens when two vehicles do not cross, but both try to leave the intersection at the same output port simultaneously. For example the sequences $Ni1 \rightarrow Eo1$ and $Si1 \rightarrow Eo1$ conflict if attempted simultaneously. The failure of the controller to handle this case is not catastrophic. In real traffic systems these conflicts are often resolved by imposing driving rules such as “right of way”. Such rules are more appropriately modeled in the vehicle behaviors.

The global behavior proceeds as follows:

1. The leading vehicles at each enabled input port on an intersection move concurrently to their specified output ports. In other words, the vehicle data flow through the intersection and the advance of the automata representing vehicles at enabled ports are performed synchronously and simultaneously for all waiting vehicles.
2. All controllers advance one transition to the next state.
3. All vehicles at output ports move simultaneously to their next input port, by moving along the adjoining street. They are queued at the next input port.
4. Repeat forever.

We could describe the entire semantics directly (and fairly simply in this case) using AsmL constructs in one large abstract state machine. The difficulty arises in carefully structuring those semantics for analysis and clarity. Instead, we define the behavior as a composition of semantic units with familiar properties (dataflow and finite state machines). Note that in the case of the controllers and the vehicles, concepts from each metamodel are bound differently to the finite-state machine behaviors.

Chen's work on the composition of semantic units [6] gives a detailed look at this type of approach. The concepts here differ only slightly, as our framework gives the designer a specification language that can be used to directly relate concepts within models and their associated semantic units rather than continuing to build up semantics hierarchically. Both techniques offer distinct but different advantages as well as limitations. The differences should not be seen as competitive, rather, as an exploration of possibilities and the issues surrounding them. The present approach is equally compatible with semantic units defined individually or through composition.

A global scheduler captures the semantics of the composed languages. The global scheduler must be inherently concurrent in order to faithfully model concurrent situations. In our toy modeling language some concurrent situations conflict while up to four vehicles can move safely and concurrently through the intersection in other situations (e.g. if everyone turns right). The global scheduler represents the semantics of the composition, and it must respect equivalences between models as well as sequential dependencies.

Consider the block diagram of Fig. 4. This represents the spirit of the interaction models and the consistent interface approach used in Ptolemy. The scheduler queries each model for possible actions and then selects appropriate actions and performs updates according to the semantics. Here the interfaces are the same for each model. Each model has an associated set of actions, and the local semantics determine how many of those actions may be invoked at the next step for each model. We require a way to bind the local execution interfaces to the metamodel concepts and a language to describe the behavior of the global scheduler in terms of those execution interfaces.

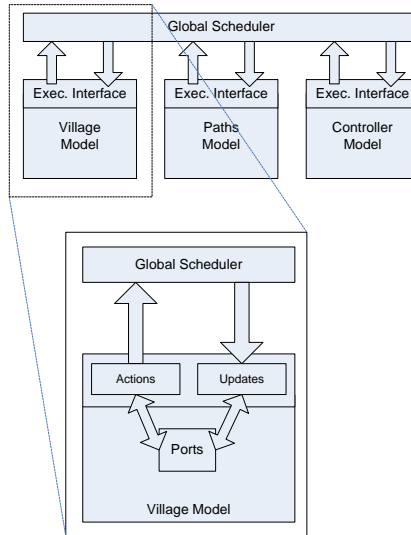


Fig. 4. Interfaces from the global scheduler to the individual models of computation. For the village model, the action and update interfaces deal only with Ports.

4.0 Behavioral Interfaces and Global Interactions

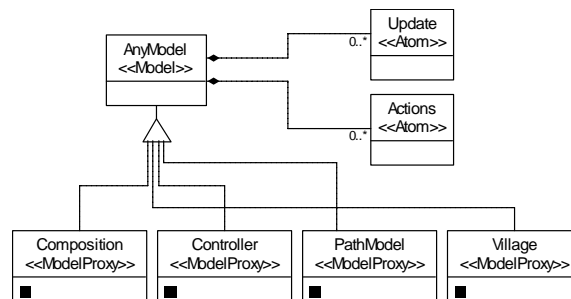


Fig. 5. Common interface for all model types in the village DSML.

We define a metamodel which allows each model to have an action request interface and a state update interface, as shown in Fig. 5. In a true industrial-strength DSML, these interfaces would be defined in a separate aspect in greater detail and carefully bound to model concepts. Here we steer towards simplicity and defer the association with model concepts to the semantic anchoring transformations and the AsmL data model for the various models of computation [4]. We also require all models to provide these interfaces, so we may add a model constraint (not shown here) to enforce their existence in the design environment. These two interfaces are analogous to function calls which could be made on each of the models.

The language for the global scheduler is shown in Fig. 6. A designer uses the instantiated action and update interfaces for each model and specifies the temporal ordering of the “calls” made. Dependencies are strict, sequentially ordered relations. The two synchronous relations are calls and barriers. Calls operate analogously to synchronous function calls in most familiar programming languages – the execution

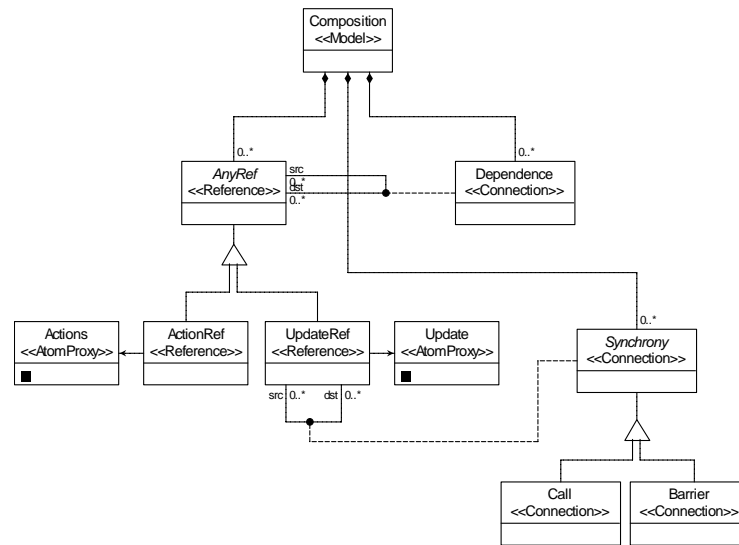


Fig. 6. Composition language for describing schedules of actions and updates.

of one call will occur completely within the invocation of the other. The language could easily be expanded to separate calls and returns, but that is beyond the scope of this simple illustration. A barrier is analogous to an explicit inter-process synchronization mechanism where all participants are required to arrive at a certain point before proceeding.

5.0 Example

For illustration we will show a sample model in this DSML. Consider a very simple village having only two intersections and one internal street (all other streets lead out of the village). The intersection ports are named for their cardinal direction (N,S,E, or W), whether input or output (i or o), and then the intersection number (1,2,...). For example, the east output of intersection one is named Eo1. Fig. 7 shows the village.

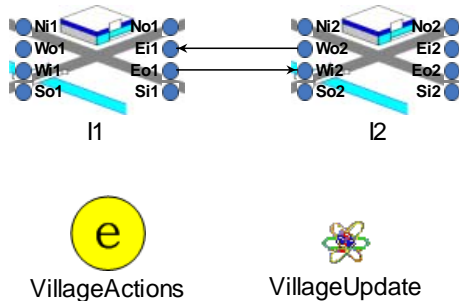


Fig. 7. Sample village.

The vehicles and their travel paths through the village are shown in Fig. 8. This sample corresponds to a potential conflict if both vehicles try to cross intersection two concurrently without a controller. This is an example of event conflict – the two event sequences ($Wi2 \rightarrow Eo2$ and $Ni2 \rightarrow So2$) conflict if taken concurrently. Since the behavior of the two vehicles is independent, there are many, many execution sequences where this will not happen.

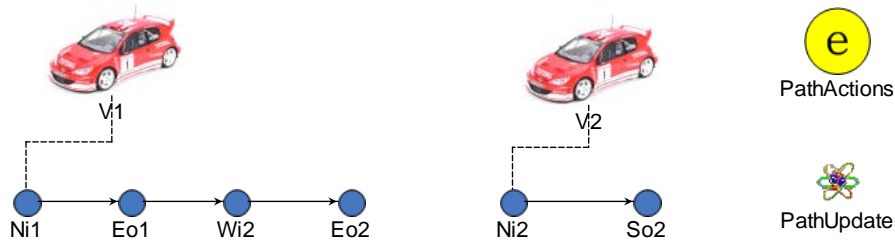


Fig. 8. Model of vehicle paths through the village.

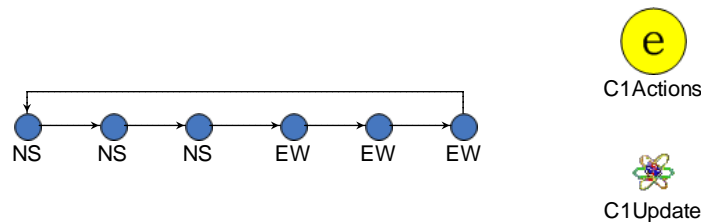


Fig. 9. Traffic controller state machine. Transitions are taken on global time ticks.

The two controllers are identical (Fig. 9). Each allows three cars to pass north-south (though it could be up to six cars if the intersection is busy in both directions), then toggles to east-west for three cycles. The cycle repeats indefinitely. Note that time triggers the change of state (there are no guards here), rather than allowing vehicles to trigger the change. In other words, the controllers advance even if no vehicles arrive.

Fig. 10 depicts the global scheduler. Solid arrows are dependencies, dashed arrows are synchronous calls, and dashed double-ended connectors are time barriers. In this case the barriers are redundant because of the explicit dependencies back to the first invocation of VillageActions. They are included here to help visually delineate the end of the schedule cycle.

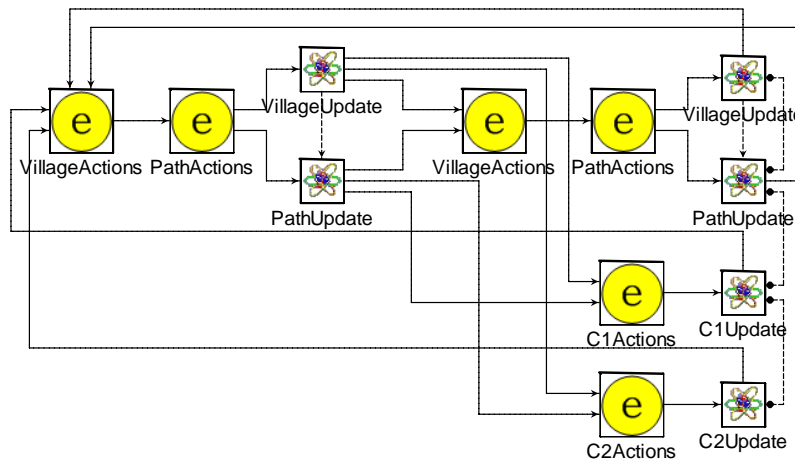


Fig. 10. Global scheduler specification for the village modeling language.

Were we to automatically generate ASML code for this scheduler, it might look something like the following:

```
// Utility functions
VillagesAndPaths()
  step
    a1 = village.actions()
  step
    a2 = paths.actions()
  step
    village.update( a1, paths.update( a2 ) )
```

```
C1()
  step
    a1 = c1.actions()
  step
    c1.update( a1 )
```

```
C2()  
  step  
    a1 = c2.actions()  
  step  
    c2.update( a1 )
```

```
// Global Schedule Single Cycle  
Cycle()  
  step  
    VillagesAndPaths()  
  step  
    VillagesAndPaths()  
    C1()  
    C2()
```

The AsmL schedule attempts to faithfully represent the concurrent nature of our toy modeling language. The independent actions for the villages and paths and for the two controllers all execute in the same step, capturing the independence of the three sets of behaviors.

6.0 Conclusions and Future Work

We have offered here a sketch of how behavioral descriptions in semantic units can be composed to specify behaviors for a DSML in a flexible way. This general approach seems to facilitate the handling of concurrent behaviors, but it may be too expressive when compared with the hierarchical build-up of semantic units presented earlier. In any case, many concepts from prior work can be captured and integrated without significant changes in the modeling tools and techniques.

Future work in this area includes the obvious extension of these concepts to more detailed examples, allowing us to flesh out the missing pieces in semantic unit composition. Of particular interest is the extension to timed models for real-time or hybrid systems. Other interesting directions include automated synthesis and analysis of denotational and axiomatic models directly from our semantic specifications. This may include partial orders models for verification.

References

1. Microsoft Research Foundations of Software Engineering
<http://research.microsoft.com/fse/asml/>

2. Egon Börger, Roark Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag. Berlin. 2003.
3. Kai Chen, Janos Sztipanovits, Sandeep Neema, Matthew Emerson and Sherif Abdelwahed, "Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages," *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, September 19-22, 2005, Jersey City, New Jersey, USA.
4. Kai Chen, Janos Sztipanovits, Serif Abdelwahed and Ethan Jackson, "Semantic Anchoring with Model Transformations", *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, November 7-10, 2005, Nuremberg, Germany, LNCS Vol. 3748.
5. Kai Chen, Janos Sztipanovits and Sherif Abdelwahed, "A Semantic Unit for Timed Automata Based Modeling Languages", *In Proceedings 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pages 347-360, San Jose, California, April, 2006.
6. Kai Chen, Janos Sztipanovits and Sandeep Neema, "Compositional Specification of Behavioral Semantics," *Submitted to the Sixth ACM International Conference on Embedded Software (EMSOFT'06)*.
7. Gregor Gössler, Joseph Sifakis. *Composition for Component-Based Modeling*. Proceedings of FMCO'02, November 2002, Leiden, the Netherlands, LNCS 2852, pages 443-466.
8. Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2003. Latest version available
<http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/>
9. Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. 1992. 1999 edition available
http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html
10. Vaughan Pratt. *Modeling Concurrency with Partial Orders*. *Int. Journal of Parallel Programming*. 15. 1, c. Nov. 1986.
11. Vaughan Pratt. *Transition and Cancellation in Concurrency and Branching Time*. *Math. Struct in Comp. Sci.* 13:4, p. 485-529. Aug. 2003.
12. Ptolemy II – heterogeneous modeling and design.
<http://ptolemy.berkeley.edu/ptolemyII/index.htm>
13. Winskel, G. *Introduction to Event Structures. Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*. LNCS 354, May/June 1988. p. 364-397.