A Case Study on Semantic Unit Composition

Kai Chen[1], Janos Sztipanovits[2] and Sandeep Neema[2]
[1]*Motorola Labs, Schaumburg, IL, 60196, USA*
Kai.chen@motorola.com
[2]*Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, 37205, USA*
{Janos.sztipanovits, sandeep.neema}@vanderbilt.edu

Abstract

In previous work we have discussed a semantic anchoring framework that enables the semantic specification of Domain-Specific Modeling Languages by specifying semantic anchoring rules to predefined semantic units. This framework is further extended to support heterogeneous systems by developing a method for the composition of semantic units. In this paper, we explain the semantic unit composition through a case study.

1. Introduction

In [2] [3], we have proposed a semantic anchoring framework (SAF) for Domain-Specific Modeling Languages (DSMLs) [1]] semantic specification. The framework includes a set of well-defined "semantic units" (SUs) that capture the behavioral semantics of basic behavioral categories using Abstract State Machines (ASM) [4] [7] as an underlying formal framework. The semantics of a DSML is defined by specifying the transformation rules between the abstract syntax metamodel of the DSML and that of a selected SU. Furthermore, we extend the SAF to address the impact of system heterogeneity by developing a method to specify DSML semantics as the composition of SUs [5]. In this paper, we explain semantic unit composition (SUC) in details using an industrial strength DSML – TNSM [6] as a case study.

The organization of this paper is the following: Section 2 introduces the core ideas of SUC. In Section 3, 4, 5 and 6 we explain the SUC in details using a case study. Our conclusion is in section 7.

2. Semantic Unit Composition

In the SAF, we define a finite set of SUs, which capture the semantics of basic behavioral and interaction categories. If the semantics of a DSML can be directly anchored to one of these basic categories, its semantics can be defined by simply specifying the model transformation rules between the metamodel of the DSML and the Abstract Data Model (ADM) of the SU [2] [3]. However, in heterogeneous systems, the semantics is not always fully captured by a predefined SU. If the semantics is specified from scratch it is not only expensive but we loose the advantages of anchoring the semantics to (a set of) common and well-established SUs. This is not only loosing reusability of previous efforts, but has negative consequences on our ability to relate semantics of DSMLs to each other and to guide language designers to use well understood and safe behavioral and interaction semantic "building blocks" as well.

Our proposed solution is to define semantics for heterogeneous DSMLs as the composition of semantic units. If the composed semantics specifies a behavior which is frequently used in system design, the resulting semantics can be considered a *derived SU*, which is built on *primary SUs*, and could be offered up as one of the set of SUs for future anchoring efforts. Note that *primary SUs* refer to the SUs that capture the semantics of the *basic behavioral categories*, such as Finite State Machine, Timed Automata and Hybrid Automata.

Mathematically, a SU specification can be represented as a 2-tuple $<A, R>$, where A is an ADM specifying the abstract syntax of the SU and R represents a set of Operations and Transition Rules. We use $M = I(A)$ to denote the set of all instances of A. Then, each $m \in M$ is a well formed Data Model defined by the A and R specifies the behavior of each $m \in M$. The behavior in ASM is modeled by a sequence of *steps*, where a Step in a given state includes the execution *simultaneously* of all Rules whose guard conditions are true [4]. Since ASM states are mathematical structures (*sets* with basic operations and predicates), it is easy to integrate ADMs and Rules. The integrated tool suite ensures that the behavior of domain models defined in a DSML is simulated

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

according to their "reference semantics" by automatically transforming them into AsmL Data Models using the transformation rules.

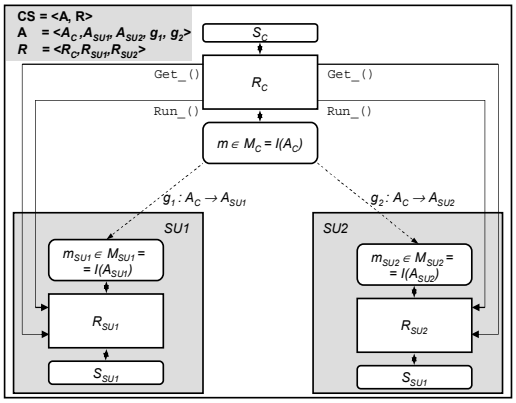

Figure 1. A graphical representation for SUC

We model SUC as *structural* and *behavioral compositions*. An ASM instance includes an *m* data model, the *R* rule set and the *S* dynamic state variables updated during runs. The structural composition defines relationships among selected elements of ADMs using partial maps. In Figure 1, we demonstrate semantic composition with two semantic units, SU1 and SU2. The composed semantics is also represented as a 2-tuple $<A, R>$. The structural composition yields the composed ADM $A = <A_C, A_{SU1}, A_{SU2}, g_1, g_2>$, where g_1, g_2 are the partial maps between concepts in A_C, A_{SU1}, and A_{SU2}.

Behavioral composition is completed by the R_C set of rules that together with R_{SU1} and R_{SU2} form the R rule set for the composed semantics. The role of the R_C set of rules is to receive the possible sets of actions that can be offered by the embedded semantic units using the Get(…) rules, to restrict these sets according to the interactions created by the structural composition and to send back selected subset of actions through the Run(…) rules to complete their next step. The executable actions are represented as partial orders above the set of actions.

3. STNSM Overview

TNSM has been developed by General Motors Research to specify vehicle motion control (VMC) software [6]. To focus on the core ideas of SUC, we introduce a simplified TNSM, called STNSM, which only includes those modeling constructs that determine the core behavioral semantics of TNSM. The full semantic specifications can be downloaded from [8].

A STNSM model is a synchronous reactive system including a set of components communicating through event channels and data channels. In each computation cycle, a STNSM system is first activated by an incoming event; this event is then propagated through event channels and activates internal components; the reaction of internal components may produce additional events; new generated events will continue the propagation and activation cycle until conclusion. According to the synchrony assumption, a computation cycle will be finished before the next incoming event triggers a new reaction.

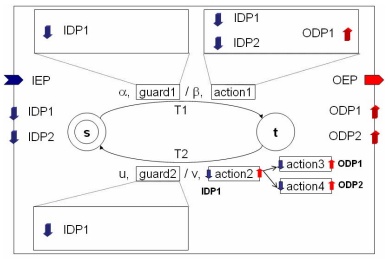

Figure 2. A simple STNSM component model

A STNSM component is an FSM-based model. We use a simple component model shown in Figure 2 as an example to explain the structure and the behavior of STNSM components. The component communicates with other components through *ports*, including a single *input event port* (IEP), an *output event port* (OEP), two *input data ports* (IDP1 and IDP2) and two *output data ports* (ODP1 and ODP2). A component also includes an FSM, where *transitions* are labeled with a *trigger event*, a *guard*, an *output event* and set of *actions*. Guards and actions are *computational functions* within the component and receive their input data through *input data ports.* The execution of an action (a function) may produce new data, while the execution of a guard only returns a Boolean value for the true or false evaluation.

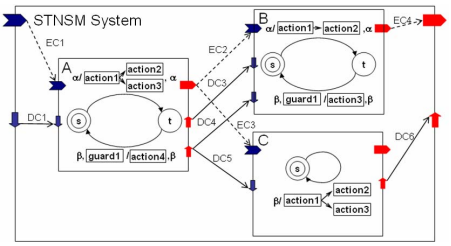

Figure 3. A simple STNSM system model

A STNSM system consists of a set of components, event channels, data channels, an input and an output event port, and a set of input and output data ports. Figure 3 presents a simple STNSM system model, including three components A, B and C. Event

channels are represented as dashed lines and data channels are shown as concrete lines.

4. Primary Semantic Units Used

In the following section we briefly elaborate the primary SUs, FSM-SU and SDF-SU, that we use to compose the semantics of, first STNSM Components, and then STNSM Systems.

4.1. FSM-SU Specification

The specification contains two parts: an ADM $A_{FSM\text{-}SU}$ and Operations and Transformation Rules $R_{FSM\text{-}SU}$ on the data structures defined in $A_{FSM\text{-}SU}$. The AsmL abstract class FSM prescribes the top-level structure of a FSM. All the abstract members of FSM are further specified by a concrete FSM, which is an instance of the Abstract State Model. (see detailed examples in [3])

```
structure Event
  eventType as String
class State
  initial    as Boolean
  var active as Boolean = false
class Transition
abstract class FSM
  abstract property states           as Set of State
    get
  abstract property transitions      as Set of Transition
    get
  abstract property outTransitions   as Map of
    <State, Set of Transition>
    get
  abstract property dstState as Map of <Transition, State>
    get
  abstract property triggerEventType as Map of
    <Transition, String>
    get
  abstract property outputEventType  as Map of
    <Transition, String>
    get
```

The behavioral semantics of FSM-SU is specified as a set of AsmL rules. The rule *Run* specifies the top-level system reaction of a FSM when it receives an event. Note that the '?' modifier after *Event* means the return from the *Run* rule may be either an event or an AsmL *null* value.

```
abstract class FSM
  Run (e as Event) as Event?
    step
      let CS as State = GetCurrentState ()
    step
      let enabledTs as Set of Transition = {t | t in
        outTransitions (CS) where e.eventType =
        triggerEventType(t)}
    step
      if Size (enabledTs) >= 1 then
        choose t in enabledTs
          step
            CS.active := false
          step
            dstState(t).active := true
          step
            if t in me.outputEventType then
              return Event(outputEventType(t))
            else
              return null
      else
        return null
```

4.2. SDF-SU Specification

The AsmL specification of the ADM $A_{SDF\text{-}SU}$ is shown below. Token is defined as an AsmL structure to package data using the AsmL construct *case*. Port and Channel are defined as first-class types. The Boolean attribute exist of a port indicates whether the port has a valid data token. When all the input ports of a node have valid data tokens, the node is enabled to fire. In the specification, *Fire* is an abstract function, which will be overridden by a concrete node with a computational function. The AsmL abstract class SDF captures the top-level structure of a model.

```
structure Value
  case IntValue
    v as Integer
  case DoubleValue
    v as Double
  case BoolValue
    v as Boolean
structure Token
  value as Value?
class Port
  var token as Token   = Token (null)
  var exist as Boolean = false
class Channel
  srcPort as Port
  dstPort as Port
abstract class Node
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
  abstract Fire ()
abstract class SDF
  abstract property nodes       as Set of Node
    get
  abstract property channels    as Set of Channel
    get
  abstract property inputPorts  as Seq of Port
    get
  abstract property outputPorts as Seq of Port
    get
```

The operational rule *Run* specifies the steps it takes to execute a set of nodes. This rule can be considered as a composition interface for SDF-SU. The rule non-deterministically chooses an enabled node from the set of enabled nodes (returned by the operational rule *GetEnabledNodes*) and fires it. The execution of a node consumes the data tokens in all input ports of the node and produce tokens to all output ports as well. An error is reported if there are no enabled nodes in the set while the set is not empty.

```
abstract class SDF
  Run (ns as Set of Node)
    step while Size(ns) <> 0
      choose n in ns where n in GetEnabledNodes ()
        remove n from ns
        Fire (n)
      ifnone
        error ("Some Nodes are not enabled to fire.")
```

5. Semantic Specification for STNSM Components

The behaviors of individual STNSM components can be divided into two different behavioral aspects: the FSM-based behaviors expressing reactions to

events and the SDF-based behaviors controlling the execution of computational functions (actions and guards). In this section, we formally specify the behavioral semantics of STNSM components as the composition of two primary semantic units: FSM-SU and SDF-SU. The compositional semantics specification consists of two parts: (1) an ADM defining the structural composition $<A_C, A_{FSM\text{-}SU}, A_{SDF\text{-}SU}, g_1, g_2>$, where $g_1: A_C \rightarrow A_{FSM\text{-}SU}$ and $g_2: A_C \rightarrow A_{SDF\text{-}SU}$ are structural relation maps; and (2) Operations and Transformation Rules specifying the behavioral composition $<R_C, R_{FSM\text{-}SU}, R_{SDF\text{-}SU}>$.

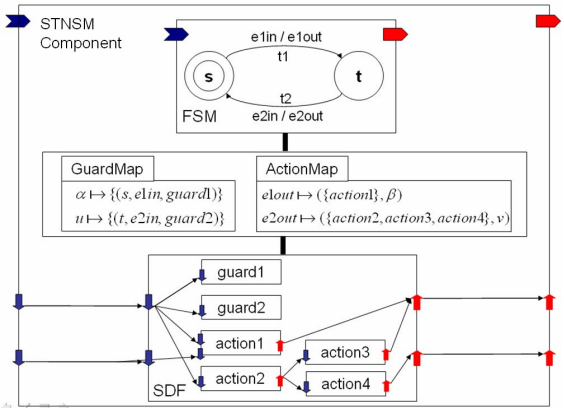

Figure 4. A compositional structure of the STNSM component shown in Figure 2.

5.1. Structural Composition

The structural composition defines mapping from elements in the ADM of the composed SU to elements in FSM-SU and those in SDF-SU. Figure 4 shows the role of FSM-SU and SDF-SU in the STNSM component model by restructuring the example in Figure 2. In the modified structure, the FSM model controls the event-related behaviors, while the SDF model takes charge of the data-related computations. Comparing Figure 2 and 4, we can find that the overall structure of the FSM model closely matches that of the original STNSM component, except for events, guards and actions. The trigger events and the output events in the FSM model are renamed. The guards and actions are represented as nodes in the SDF model. The relationships between the FSM model and the SDF model are specified by two maps: *GuardMap* and *ActionMap*. In this section, we only briefly explain how these two maps help to relate the FSM model with the SDF model.

The new compositional structure is built in a way that each transition in the original component is decomposed into three parts: a transition in the FSM model, a node representing the guard and a node representing the action in the SDF model. In the original component, a transition can be unambiguously located by the combination of the source state, the trigger event, and the guard. In the compositional structure, the information can be expressed by a 3-tuple *(s, e, n)*, where *s* refers a state in the FSM model; *e* is a local trigger event in the FSM model; and *n* represents a node in the SDF model. When a component receives an event, this event is a global event and will not be directly forwarded to the FSM model. The *GuardMap* maps this global event to a set of 3-tuples, each tuple referring to a transition in the original component whose trigger event matches this global event. Using the example in Figure 2 again, the event α is the trigger event only for the transition *T1*. In the compositional structure as shown in Figure 4, the *T1* transition is decomposed into the *t1* transition in the FSM model, whose source state is *s* and trigger event is *e1in*, and the *guard1* and *action1* node in the SDF model. As a result, *GuardMap* assigns the event α to the set *{(s, e1in, guard1)}*.

```
class EventPort
  var evnt   as Event  = Event ("")
  var exist  as Boolean = false
abstract class Component
  abstract property inPort    as EventPort
    get
  abstract property outPort   as EventPort
    get
  abstract property GuardMap  as Map of <String,
    Set of(String, String, Node?)>
    get
  abstract property ActionMap as Map of <String,
    (Set of Node, String?)>
    get
  abstract property fsm as FSM
    get
  abstract property sdf as SDF
    get
```

5.2. Behavioral Composition

In essence, the behavioral composition specifies the rules R_C, which is akin to a component-level controller (or scheduler) that orchestrates the executions and interactions of the FSM model and the SDF model.

The execution of a transition in the original STNSM component can be decomposed into a three-step process: (1) the evaluation of the guard functions for all outgoing transitions from the current state as nodes in the SDF model; (2) the selection of an enabled transition in the FSM model; and (3) the execution of actions of the transition as nodes in the SDF model. The three steps are related to each other by the maps *GuardMap* and *ActionMap*. The output event produced by the execution of a transition in the FSM model is a local event. *ActionMap* maps it to a 2-tuple *({n}, e)*, where *{n}* refers to a set of nodes (actions) in the SDF model and *e* refers to a global output event that will be propagated out of the component. For instance, the execution of the *t2* transition of the FSM model in

Figure 4 generates a local event *e2out*. Since the *t2* transition corresponds to the *T2* transition in the original component (Figure 2), which is attached with actions, *action2*, *action3* and *action4*, and an output event *v*, the *ActionMap* maps the local event *e1out* to a 2-tuple *({action2, action3, action4}, v)* accordingly.

The rules verbalized above are specified in AsmL as Operations and Transition Rules. The operational rule *Run* of *Component* specifies the top-level component operations as a sequence of *updates*. The rule first consumes the event in the port and checks whether this event triggers further updates in the component. If the event does, the rule *MapToLocalInputEvent* returns the corresponding local event used to trigger the FSM model; if not, a *null* value is returned and the reaction is completed. If a valid local event is returned, it activates the FSM model. The reaction of the FSM model returns a local output event. If the STNSM component produces an output event in this reaction, the rule *MapToGlobalOutputEvent* maps the local event to the global output even, which is then stored in the output port of the component.

```
abstract class Component
  Run ()
    require inPort.exist
    step
      inPort.exist := false
      let localEvent as Event? =
        MapToLocalInputEvent (inPort.evnt)
    step
      if localEvent <> null then
        step let e as Event? = fsm.Run (localEvent)
        step
          let globalEvent as Event?=MapToGlobalOutputEvent(e)
        step
          if globalEvent <> null then
            outPort.evnt := globalEvent
            outPort.exist := true
```

The semantics of STNSM components is defined as the composition of the two semantic units: FSM-SU and SDF-SU. We observe that this behavioral semantics specification is not limited to the STNSM components. It actually specifies the semantics of a common behavioral category that captures the reactive computation behaviors. Therefore, we can consider the semantic specification for STNSM components as a new derived semantic unit, called Action Automaton Semantic Unit (AA-SU). We leverage this AA-SU in the following section to compositionally specify the semantics of STNSM Systems.

6. Semantic Specification for STNSM Systems

The semantics of STNSM systems is defined as the composition of AA-SU and SDF-SU. The semantic specification for STNSM includes: (1) an ADM defining the structural composition $<A_C, A_{AA\text{-}SU}, A_{SDF\text{-}SU}, g_1, g_2>$, where g_1: $A_C \rightarrow A_{AA\text{-}SU}$, and g_2: $A_C \rightarrow A_{SDF\text{-}SU}$ are structural relation maps; and (2) Operations and Transformation Rules specifying the behavioral composition $<R_C, R_{AA\text{-}SU}, R_{SDF\text{-}SU}>$.

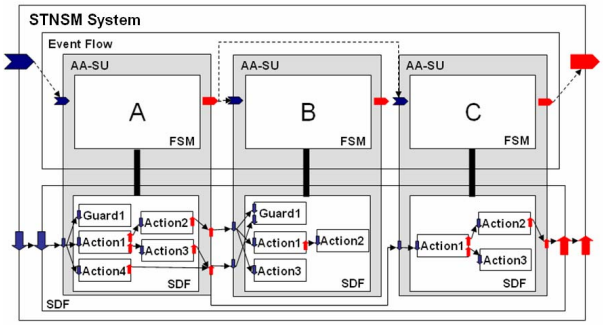

Figure 5. The compositional structure of the STNSM system originally shown in Figure 3

6.1. Structural Composition

The structural composition defines the communication relationships among components, in terms of an event flow and a data flow. As it is shown in Figure 5, we reuse again the SDF-SU to model the interaction semantics for the data flow. It is important to note that due to the integration with the FSM sections, always only a subset of the SDF nodes is involved in a reaction of the STNSM system. Figure 5 presents the role of the AA-SU, SDF-SU and the event flow interactions in the STNSM system model by restructuring the example in Figure 3. This new structure gives a much clearer expression for the control dependency among components and the data dependency among computational functions (actions and guards).

```
class EventChannel
  id      as String
  srcPort as EventPort
  dstPort as EventPort
abstract class System
  abstract property inPort     as EventPort
    get
  abstract property outPort    as EventPort
    get
  abstract property components as Set of Component
    get
  abstract property channels   as Set of EventChannel
    get
  abstract property sdf        as SDF
    get
```

The AsmL abstract class *System* captures the top-level structure of a STNSM system. The abstract property *components* is a set holding all components in a system. The control dependency among components is expressed by a set of event channels contained in the abstract property *channels*. The data dependency among computational functions is described by a SDF model. Each component has a reference to this SDF model. The relationship between a component and the SDF model is defined by the AA-SU (e.g. the abstract

property *GuardMap* and *ActionMap* in the class *Component*).

6.2. Behavioral Composition

The behavioral composition for the STNSM system defines a system-level controller (or scheduler) that controls the executions and the order of the executions of components, event channels and the SDF model. The operational rule *Runt* of *System* specifies the top-level system operations as a sequence of *updates*. Firstly, the rule propagates the event in the input event port of the system along all the connected event channels to the destination ports referring to the input event ports of components. In the meantime, the operational rule *Initialize*, defined in the SDF-SU, propagates the data tokens in the input ports of the SDF model along the connected data channels to the destination ports that refer to the input ports of nodes. The next step is to keep running until the operations inside the step cause no further state updates in the ASM (*fixpoint*).

```
abstract class System
  Run ()
    require inPort.exist
    step
      forall c in me.channels where c.srcPort.exist
        c.dstPort.evnt := c.srcPort.evnt
        c.srcPort.exist := false
        c.dstPort.exist := true
      sdf.Initialize ()
    step until fixpoint
      step
        forall comp in me.components
          where comp.inPort.exist
          comp.Run ()
      step
        forall c in me.channels where c.srcPort.exist
          c.dstPort.evnt := c.srcPort.evnt
          c.dstPort.exist := true
          c.srcPort.exist := false
    step
      sdf.ClearPorts ()
```

Within the loop, the rule first activates all the components who receive an event. The reactions of these components then produce new events. If new events are produced, the rule propagates them to the destination components and continues the loop; otherwise, the loop is stopped. Finally, the rule *ClearPorts* defined in SDF-SU is utilized to clear all the input data ports in the SDF model because the STNSM system does not store the data generated in the last computation cycle.

This behavioral semantics is actually not unique to STNSM. Rather, it captures the common behavior of event-driven synchronous reactive systems. Therefore, we can also consider the semantic specification for STNSM as a new derived SU for event-driven synchronous reactive systems. Details of the specification clearly demonstrates the similarities between the semantics of STNSM and well known event-driven synchronous reactive systems and opens up the possibility of utilizing a rich variety of analytical techniques that have been developed in that domain.

7. Conclusion

Compositional semantic specification is a necessary step for making DSMLs semantically precise and practical. The proposed approach builds on a large body of work on ASM [4] [7] and on our earlier work on the semantic anchoring methodology [2] [3]. As a future step we will continue the construction of a library of primary semantic units and will move toward increased automation in semantic unit composition.

9. References

[1] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model Integrated development of embedded software. *Proceedings of the IEEE*, volume 91, pages 145–164, 2003.

[2] Chen K., Sztipanovits J., Neema S., Emerson M., Abdelwahed S. Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages, In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, pages 35-44, New Jersey, September, 2005.

[3] Chen K., Sztipanovits J., Abdelwahed S., Jackson E. Semantic Anchoring with Model Transformations. In *Proceedings of European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA)*, Nuremberg, Germany, November, 2005. Lecture Notes in Computer Science, vol. 3748. pages 115-129.

[4] E. Boerger and R. Staerk. *Abstract State Machines: A Method for HighLevel System Design and Analysis*. Springer, 2003.

[5] Chen K., Sztipanovits J. and Neema S. Compositional Specification of Behavioral Semantics. Accepted by *Proceedings of Design, Automation, and Test in Europe 2007*, Nice, France, April, 2007.

[6] S. Birla, S. Wang, S. Neema, and T. Saxena. Addressing cross-tool semantic ambiguities in behavior modeling for vehicle motion control. In *Automotive Software Workshop 2006*, San Diego, CA, April 2006.

[7] The Abstract State Machine Language. www.research.microsoft.com/fse/asml..

[8] The Semantic Anchoring Tool Suite. www.isis.vanderbilt.edu/SAT.