Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Validating Transformations for Semantic Anchoring

David Lindecker[a] Gabor Simko[a] Tihamer Levendovszky[a]
Istvan Madari[a] Janos Sztipanovits[a]

a. Vanderbilt University, Nashville, TN, USA
{david.a.lindecker,gabor.simko,tihamer.levendovszky,istvan.madari,janos.sztipanovits}@vanderbilt.edu

Abstract Making Domain-Specific Modeling Languages a part of a tool chain, a part of a proven development process, or the subject of verification cannot be achieved without the precise specification of the language and the models expressed in it. Defining formal semantics for modeling languages is a widely accepted solution to this problem. We have developed methods, techniques and processes to provide a systematic mapping – which we call semantic anchoring – that support the scaling of these formal definitions to large modeling languages. Although a semantic mapping is a definition and behaves as a set of axioms for formal verification, we argue that a semantic mapping can conflict with the informal intentions of the language designer, resulting in a counterintuitive DSML, and should therefore be validated. This paper proposes a solution that involves formalizing the language designer's intentions about the semantic mapping and validating the consistency between the two by applying model finding techniques.

1 Introduction

Domain-Specific Modeling Languages (DSML) distance engineers from low-level implementation details by creating high-level abstractions that are familiar to experts in the corresponding field. Naturally, when designing these DSMLs, we must specify what these abstractions mean. Sometimes, informal or implicit descriptions will suffice. In domains with high-confidence requirements (e.g. flight systems, medical devices) it is common to specify precise and formal semantics for the language.

A semantics should contain the language designer's intentions about the meaning of the language's abstractions. This can be accomplished by mapping the language elements to structures which describe their meaning. We have found that such a semantic mapping can sometimes become large and complex, allowing it to contain subtle and obfuscated inconsistencies with the designer's intentions, resulting in a

David Lindecker, Gabor Simko, Tihamer Levendovszky, Istvan Madari, Janos Sztipanovits. *Validating Transformations for Semantic Anchoring*. Licensed under Attribution 4.0 International (CC BY 4.0).
In Journal of Object Technology, vol. 14, no. 3, 2015, pages 2:1–25. doi:10.5381/jot.2015.14.1.a2

counterintuitive or even harmful semantics. This introduces validation requirements on the semantic mapping itself.

Our solution is to create, in addition to the semantic mapping, a more direct formalization of the language designer's intentions expressed as a set of validation axioms about the semantic mapping and then validate the semantic mapping's consistency with these axioms. We illustrate this approach with a case study on a simple statecharts [Har87] language, the semantics for which we specify via a mapping to the abstract syntax formalized in [HR07]. We use FORMULA [JS09, JSB12] as a specification language and make use of its goal-based model finding features to search for counterexamples to the consistency of the semantic mapping and its validation axioms.

The rest of the paper is organized as follows. Section 2 gives an overview of the context within which our work is framed. In Section 3 we give an introduction to the specification and analysis features of FORMULA that we use throughout the paper. We describe our process for translating visual modeling artifacts to a corresponding encoding in FORMULA in Section 4 and in Section 5 we detail the different types of semantic specifications that we deal with and how we encode them in FORMULA. Section 6 describes the axiomitization of the intentions about a semantic mapping's behavior. We present examples of our validation process in Section 7. A critical discussion of our contribution can be found in Section 8. Section 9 discusses the related literature and we discuss our conclusions in Section 10.

2 Background

The power of DSMLs lies in the flexibility stemming from the customizable infrastructure. However, the same flexibility makes it difficult to identify the interpretation of the information that an often evolving domain-specific model represents. As a database can have a schema that specifies the structure and value domains of valid data, the well-formedness rules of DSMLs are also specified by some means of language definition, typically a metamodel. It is important to note that while an integer column contains only numbers in a database table, its interpretation (e.g. that it is a social security number) is not attached to the schema or the data: it is usually encoded in the program which uses it or, even more frequently, in the mind of the user who uses the data. Similarly, a DSML metamodel/model does not inherently have an interpretation for its modeling elements, such as a box or a line: it may be implicit in model translators or understood by the human end-user.

In certain domains, such as Cyber-Physical Systems (CPS) [KS08], this impreciseness is unacceptable: formal verification tools and rigorous certification processes are built on the information stored in a DSM – its interpretation must be precise and unambiguous. In principle, the specification of the interpretation can be of any form as long as it holds these two properties. One approach is to map the DSM elements to some mathematical structure, such as equations or an automaton, or to another language with its own semantic specification. This method is referred to as assigning a *semantic mapping* to the language [HR04], and the target of the mapping is called the *semantic domain*. The semantic domain and the semantic mapping together are called the (formal) semantics of the language.

Figure 1 shows a visualization of our systematic approach to defining a formal semantics, which we refer to as *semantic anchoring*. First we have to create a formal representation of the language specification and the models. Note that conceptually

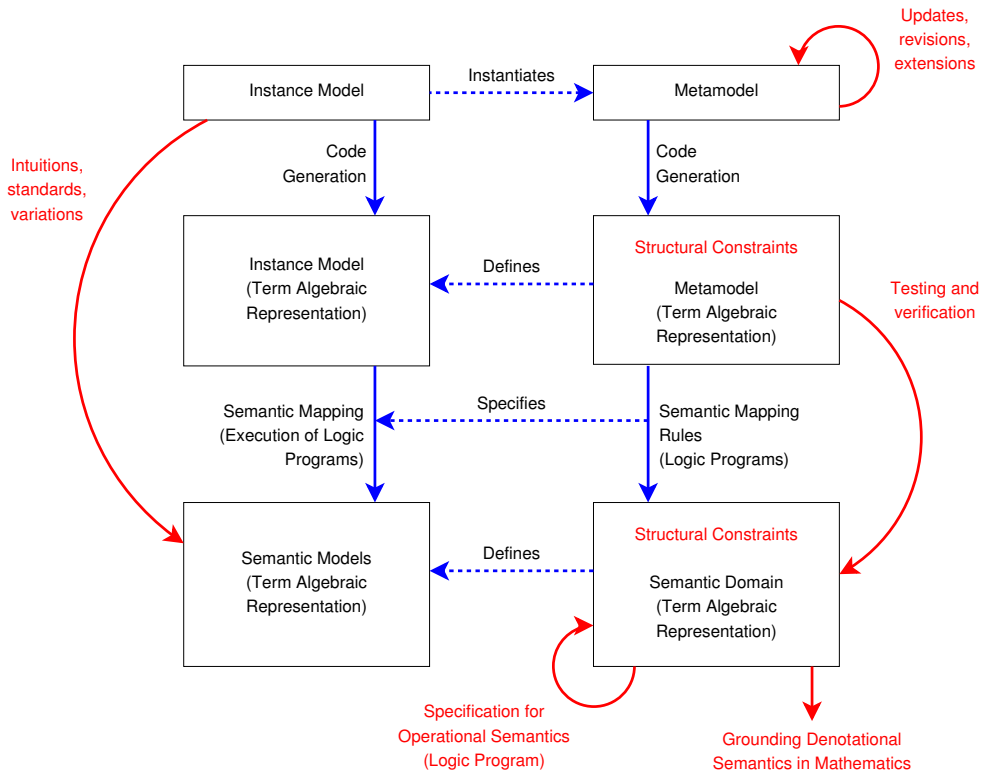


Figure 1 – Semantic Anchoring Dissected

this formal representation has nothing to do with the interpretation: it only describes formally the structure and well-formedness of the information represented, also known as the abstract syntax of the language. In the context of databases this could be the relational data model [Che76]; for DSMLs, they can be labeled graphs or certain algebraic structures such as term algebra [SB81], or certain categorical constructs (category of typed attributed graphs, [EEPT06]). In practice, however, language designers try to make sure that the modelers cannot express models that cannot be mapped to a valid semantic interpretation. Therefore, the interpretation (and thus the formal semantics) usually motivates many of the well-formedness constraints of the language. For example, we may want to ensure that language users cannot create models containing a multiplier block with a corresponding DAE system that does not have a unique solution. This is a realistic expectation – it is typically undesirable to create models that are ambiguous (have multiple interpretations) or cannot be implemented (have no interpretation). In the context of domains such as Cyber-Physical Systems, the formal representation of the language and the well-formedness constraints are referred to as *structural semantics*. Note that a structural semantics does not assign any interpretation to the language; it is in fact an abstract syntax specification motivated by semantic considerations. The name comes from modeling languages similar to Simulink, where structural semantics describes what type of block can be connected to what other type of block to create a model structure that can be simulated (implemented by code generators), while the behavioral semantics describes

how to simulate (what code to generate from) the model.

Having formalized the specification and the abstract syntax of the language, we have to create a mapping from the language to the semantic domain such that for each model of the language, a member of the semantic domain and their correspondence can be derived. This can be done in several ways and is a popular application of graph/model transformations, logic programs, and the fusion of the two. We have taken the latter approach provided by the tool FORMULA [JSB12] developed by Microsoft Research.

FORMULA has been created for the semantic mapping problem described above [JS09]. The language specification – called a domain – is expressed by term algebras, and the well-formedness constraints can be expressed by logic statements defined over the term algebras. It can also express the models of the specification, and is able to verify the well-formedness constraints, i.e. if the model conforms to its domain. The semantic mapping is expressed as transformations built on logic programs. Semantic domains can be expressed as FORMULA domains specifying the mathematical structures as described in [VP03b]. The members of the semantic domains are FORMULA models. For operational semantic domains such as automata, the execution transitions can also be defined with transformations.

When modeling languages become large, scaling the semantic anchoring technique becomes challenging. In many cases, we have found that these large modeling languages can be divided into several interrelated sublanguages. We have developed techniques for organizing structured semantic specifications for such languages, which we call a *semantic backplane*. These techniques are explained in [SLN+12].

3 FORMULA Language

FORMULA [JS09, JSB12] is a constraint-logic programming language based on abstract data types and restricted domains, making it highly applicable to the specification and analysis of DSMLs. Here we provide a brief introduction to its syntax and analysis features.

The basic organizational concepts in FORMULA are domains and transformations, each of which contain type signatures and inference rules. Intutively, a domain is a specification for a set of models and a transformation specifies a mapping from models belonging to one domain to models belonging to another domain. Practically, domains and transformations are logic programs, the input for which is a model. These programs are executed by inferring terms until a logical fixpoint is reached. FORMULA enforces stratified programs which ensures that this fixpoint is unique.

A domain's type signatures are divided, by use of the `primitive` keyword, into those which can be used to build a model's initial knowledge set and those which can be used by inference rules. Consider the following domain specification:

```
domain DirectedGraph {
  primitive Vertex ::= (index:Integer).
  [Closed(src,dst)]
  primitive Edge ::= (src:Vertex, dst:Vertex).

  Path ::= (src:Vertex, dst:Vertex).

  Path(src, dst) :-
    Edge(src, dst);
    Edge(src, v), Path(v, dst).
}
```

We have two type signatures, `Vertex` and `Edge`, which define the typing requirements for the terms constituting the initial knowledge of a `DirectedGraph` model. (e.g. `Edge` terms are composed of two `Vertex` subterms.) The `Closed` directive preceding the `Edge` signature indicates that the `src` and `dst` subterms of `Edge` terms must also be top-level terms within a model. The `Path` signature, by absence of the `primitive` keyword, defines a type for derived terms. The domain also contains a single rule, identified by the `:-` symbol, which specifies the logical conditions (right-hand side) under which `Path` terms (left-hand side) should be inferred. (The semi-colon denotes top-level alternation.)

A domain can also contain *queries*, indicated by the `:=` symbol, which are similar to rules except that the left-hand side is a named constant rather than a term instantiation. For example, we could define the following query for determining whether or not a `DirectedGraph` model contains a cycle:

```
ContainsCycle := Path(v, v).
```

The special query `conforms` is used to identify valid models. It can be specified manually to realize correctness constraints and is also populated with various constraints derived from the type signatures of the domain.

A transformation contains the same types of elements as a domain, but with important differences. All type signatures are for derived terms since transformations have no instance models. Additionally, rules are allowed to derive primitive-type terms in the transformation's output domain. These inferred terms are collected together and form the transformation's output model. The following transformation maps an input `DirectedGraph` model to a corresponding output model of the same domain with all of the edges reversed:

```
transform ReverseGraph from igraph::DirectedGraph to ograph::DirectedGraph
{
  ograph.Vertex(idx) :- igraph.Vertex(idx).
  ograph.Edge(v1,v2) :- igraph.Edge(v2,v1).
}
```

The `from igraph::DirectedGraph` fragment specifies that the transformation takes an input model of the `DirectedGraph` domain and that its type signatures can be referenced within the transformation by using the `igraph.` prefix as can be seen in the right-hand side of the rules. Similarly, the fragment `to ograph::DirectedGraph` specifies an output model of the `DirectedGraph` domain and gives places its type signatures within the `ograph` namespace.

A model in FORMULA is a collection of terms corresponding to the type signatures of precisely one domain. Note that FORMULA will process a model which does not adhere to the `conforms` query of its assigned domain. However, failure of the model to adhere to the domain's type restrictions will result in a compilation error. For example, consider the following model:

```
model m1 of DirectedGraph {
  Vertex(1)
  Edge(Vertex(1), Vertex(2))
}
```

This example is valid FORMULA, although checking the `conforms` query for the model `m1` returns false because `Vertex(2)` is not a top-level term in the model. This is because the `Closed` directive does not effect FORMULA's typing mechanism, but rather adds constraints to `conforms`.

In addition to executing domain and transformation programs on models, FORMULA supports integration with the Z3 [DMB08] SMT solver for synthesizing input

models that satisfy certain desired properties. First we create a *partial* model, such as the following:

```
partial model m2 of DirectedGraph {
  Vertex(1) Vertex(2) Vertex(3) Vertex(4)
  Edge(_,_) Edge(_,_) Edge(_,_)
}
```

The underscores indicate unspecified parts of the model. This differs from underscores which appear within rules, which are variables that are not bound to an alias. In this case we have a partial model specifying a search space over 4-vertex, 3-edge graphs. Actually, 4-vertex graphs with between 1 and 3 edges since FORMULA could complete the `Edge` terms in the same way. (Models function as sets of terms; redundant terms are ignored). There are ways to avoid this behavior, but we will not discuss them here. We have specified the vertex indices since we consider them unimportant, provided they are different, as it reduces some unnecessary complexity in the search space.

We must also select a goal query for the search. We could, for example, use the `ContainsCycle` query to get a completion which contains a cycle in it. Alternately, consider the following additional queries:

```
Connected := v1 is Vertex, v2 is Vertex, no Path(v1, v2), no Path(v2, v1).
Goal := Connected & ContainsCycle.
```

As demonstrated by the `Goal` query, in addition to the logic rule based syntax, we can also specify a query as the logical composition of other queries using the `&`, `|`, and `!` operators. In this case, `Goal` is the logical conjunction of `Connected` and `ContainsCycle`. If we use it as our goal query for the partial model `m2`, FORMULA will fail to find a valid completion. This effectively proves that no such configuration exists for the specific case of 4 vertices and up to 3 edges.

4 Translating Visual Modeling Artifacts to FORMULA

Metamodeling is the preferred method of specifying the abstract syntax of a DSML, typically paired with a query language such as OCL [OMG03] for expressing additional well-formedness constraints. In order to be useful, our approach must support languages that are specified in such a way.

We accomplish this with an automated translation from metamodels created in the Generic Modeling Environment (GME) [LBM+01] to FORMULA domains. Constraints on the language can be expressed directly within the FORMULA domain as a combination of rules and queries. As of this writing, our tool does not support automatic translation of OCL constraints, but this has been accomplished by others as described in [PP13], so there is no theoretical barrier to this. In our framework, we encode constraints in FORMULA manually, so we do not demonstrate a correspondence here.

We demonstrate this translation on a simplified statecharts language taken from CyPhyML, our Cyber-Physical Systems integration and simulation language described in [LNS+12, WNO+12]. The metamodel can be seen in Figure 2. The modeling paradigm consists of states, transitions, junctions, and a `TransStart` type which is used as the source of a transition to indicate a default transition. The `State`, `Junction`, and `TransStart` classes all derive from `TransConnector`, which can be contained by `State` objects and can be the source or destination of `Transition` objects.

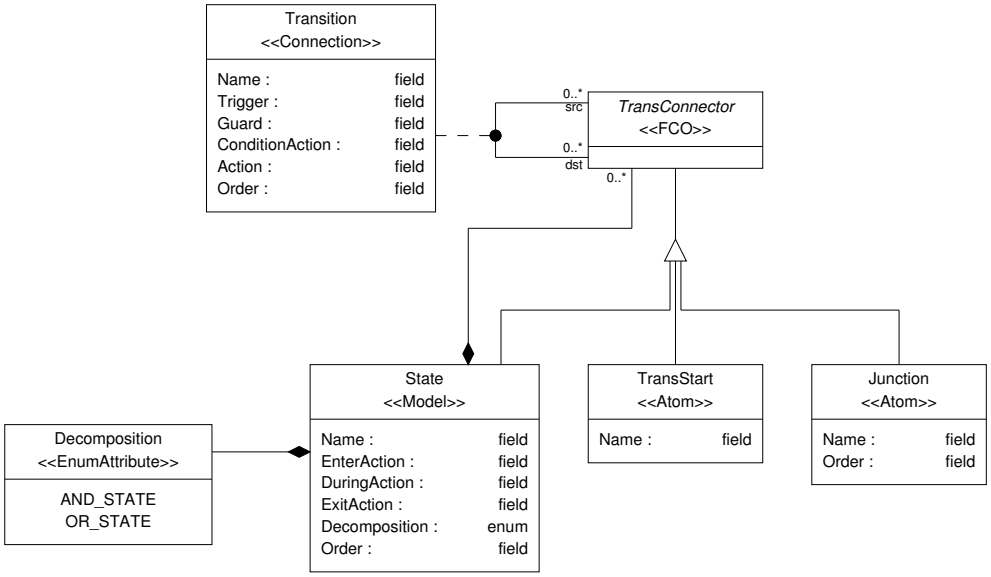


Figure 2 – GME metamodel of statecharts language

The FORMULA code generated from this metamodel can be seen in Figure 3, with some minor simplifications, explained in the following paragraphs. The attributes of the MetaGME classes are inserted into corresponding FORMULA type signatures, along with attributes that are implicit for certain types of class blocks, such as `Source` and `Destination` for `Connection` classes like `Transition`. We also generate additional type signatures for certain associations, such as `StateToTransConnectorContainment`, which clearly corresponds to the containment association between `State` and `TransConnector`. GME objects have hidden unique identifiers, and these are added to the FORMULA type signatures by the translator. We have removed them as, for this example, they add needless bulk to the code excerpts.

GME has three types of attribute: Boolean, Enum, and Field. Boolean and Enum attributes correspond to FORMULA's built-in Boolean type and enumerated type signatures, respectively. The Field type represents a generic serializable attribute and so by default we translate these to FORMULA strings, as can be seen for most of the attributes in the example. However, sometimes, this is not an ideal solution. In the case of the `Order` attribute, we would like to leverage the fact that it is an integer value without parsing it within FORMULA. We accomplish this by way of translator plug-ins which can be configured to operate on particular attributes of certain object types and generate arbitrarily complex FORMULA terms from them. In this case, we perform only a simple type conversion, converting the `Order` string to a `Natural`. Technically, the Field attribute has a data type setting supporting integers, strings, and doubles. In practice, the attribute is often left as a string in the metamodel even when the data value is always a number, as is the case with the `Order` attribute in this example.

FORMULA does not support inheritance between type signatures. We resolve this discrepancy with union types and copying of base class attributes to each derived class. If the base class is abstract, as with `TransConnector`, we can simply create a union type

```
1  domain StatechartLanguage
2  {
3    StateDecomposition ::= { AND_STATE, OR_STATE }.
4
5    TransConnector ::= State + Junction + TransStart.
6
7    primitive State ::= (Name:String, EnterAction:String, DuringAction:String,
8      ExitAction:String, Decomposition: StateDecomposition, Order:Natural).
9    [Closed(Source,Destination)]
10   primitive Transition ::= (Name:String, Source:TransConnector,
11     Destination:TransConnector, Trigger:String, Guard:String,
12     ConditionAction:String, Action:String, Order:Natural).
13   primitive Junction ::= (Name:String, Order:Natural).
14   primitive TransStart ::= (Name:String).
15   [Closed(Container,Contained)]
16   primitive StateToTransConnectorContainment ::= (
17     Container:State, Contained:TransConnector).
18 }
```

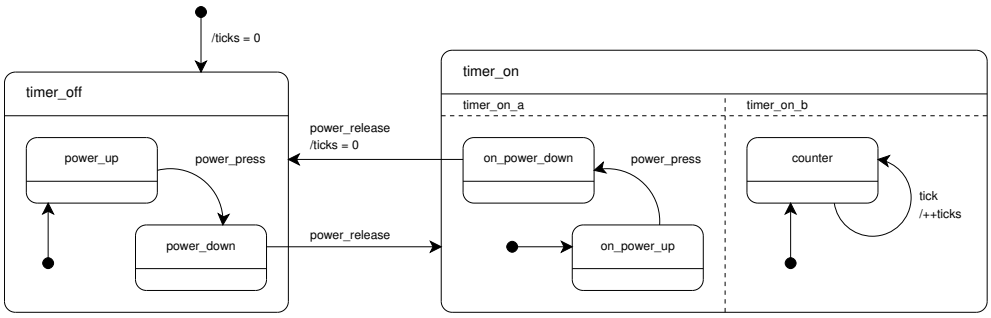
Figure 3 – Generated FORMULA domain for the Statecharts language

for it as it does not need its own concrete type signature. We have made this simplification here, although it is not the default behavior of the translator. The default behavior is to generate union types as needed, such as `Transition_Source` consisting of all types allowed to be the source of a `Transition`, or `StateToTransConnectorContainment_Contained`. This technique can result in verbose specifications, but is explicit and works well in general.

The GME-to-FORMULA translator also supports translation of GME models to FORMULA models. In fact, the translator always takes a model and its paradigm-defining metamodel as input and returns the corresponding FORMULA model and domain, but it is perhaps simpler to think of the translations separately. Figure 4a contains a visualization of a statechart model which might be built in our simplified paradigm. This model implements a simple timer mechanism which counts the number of ticks since it was started with the power button. There are two top-level states: `timer_off` and `timer_on`. The default transition into `timer_off` initializes the value of `ticks` to zero. The substates within the `timer_off` state implement a transition to the `timer_on` state when the power button is pressed and then released. The `timer_on` state contains two parallel substates, one substate which implements a similar mechanism to the one in `timer_off` and transitioning back to it, additionally resetting the value of `ticks` to zero, and one substate which contains a state-transition loop that increments the value of `ticks` every time the `tick` event is processed. There are also default transitions within the `timer_off`, `timer_on_a`, and `timer_on_b` states.

Figure 4b contains the generated FORMULA code corresponding to this model. The visual model elements become terms in the FORMULA model. For example, the `timer_on` state is translated to the term `State("timer_on", "", "", "", AND_STATE, 2)` in line 4 and the elements contained within it (e.g. its substates) are indicated by `StateToTransConnectorContainment` terms as seen in lines 12 and 20. The number at the end of `State` and `Transition` terms is used for determining evaluation order in otherwise nondeterministic conditions. Typically, it is configurable within visual modeling tools, but is not shown in the diagram, as is the case here.

As of this writing, we have made no attempt to formalize the correspondence between GME artifacts and the generated FORMULA code. We consider the two



(a) Visual representation

```
1  model m1 of StatechartLanguage {
2    ts is TransStart("ts")
3    timer_off is State("timer_off", "", "", "", OR_STATE, 1)
4    timer_on is State("timer_on", "", "", "", AND_STATE, 2)
5    power_up is State("power_up", "", "", "", OR_STATE, 1)
6    StateToTransConnectorContainment(timer_off, power_up)
7    power_down is State("power_down", "", "", "", OR_STATE, 2)
8    StateToTransConnectorContainment(timer_off, power_down)
9    timer_off_ts is TransStart("timer_off_ts")
10   StateToTransConnectorContainment(timer_off, timer_off_ts)
11   timer_on_a is State("timer_on_a", "", "", "", OR_STATE, 1)
12   StateToTransConnectorContainment(timer_on, timer_on_a)
13   timer_on_a_ts is TransStart("timer_on_a_ts")
14   StateToTransConnectorContainment(timer_on_a, timer_on_a_ts)
15   on_power_down is State("on_power_down", "", "", "", OR_STATE, 1)
16   StateToTransConnectorContainment(timer_on_a, on_power_down)
17   on_power_up is State("on_power_up", "", "", "", OR_STATE, 2)
18   StateToTransConnectorContainment(timer_on_a, on_power_up)
19   timer_on_b is State("timer_on_b", "", "", "", OR_STATE, 2)
20   StateToTransConnectorContainment(timer_on, timer_on_b)
21   timer_on_b_ts is TransStart("timer_on_b_ts")
22   StateToTransConnectorContainment(timer_on_b, timer_on_b_ts)
23   counter is State("counter", "", "", "", OR_STATE, 1)
24   StateToTransConnectorContainment(timer_on_b, counter)
25   Transition("", ts, timer_off, "", "", "", "ticks = 0", 1)
26   Transition("", timer_off_ts, power_up, "", "", "", "", 1)
27   Transition("", power_up, power_down, "power_press", "", "", "", 1)
28   Transition("", power_down, timer_on, "power_release", "", "", "", 1)
29   Transition("", timer_on_a_ts, on_power_up, "", "", "", "", 1)
30   Transition("", on_power_up, on_power_down, "power_press", "", "", "", 1)
31   Transition("", on_power_down, timer_off, "power_release", "", "", "ticks = 0", 1)
32   Transition("", timer_on_b_ts, counter, "", "", "", "", 1)
33   Transition("", counter, counter, "tick", "", "", "++ticks", 1)
34 }
```

(b) Generated FORMULA code

Figure 4 – Statechart example model

to be different representations of the same information. While some projects may require a level of certainty which demands verification of this, we leave it as a matter for future work.

5 Semantic Specifications in FORMULA

We distinguish between several different types of semantic specifications which we can define in FORMULA. The *structural semantics* is a formalization of the well-formed structures that a language model can consist of [JS09]. In FORMULA, this is realized with a domain specification, consisting of a set of type signatures for constructing models and a set of constraints for determining which model configurations are well-formed. We can generate the type signatures from a metamodel as described in Section 4. Constraints are realized with FORMULA queries (as well as logical rules if implication is needed). For example, in our statecharts language, a `TransStart` is intended to be used as the source of a transition to flag it as a default transition. It does not make sense to use it as the destination of a transition, but the specification given by the MetaGME diagram in Figure 2 does not prevent this (nor does the corresponding FORMULA domain in Figure 3). We could identify the models with such a configuration with the following simple query:

```
TransStartIsDst := Transition(_,_,dst,_,_,_,_,_), dst in TransStart.
```

We can then add this restriction to the structural semantics by including the logical negation of the above query in the domain's `conforms` query.

If a modeling language describes computation, then its semantics can be specified by pairing the language domain with the necessary constructs to describe its execution state and defining a set of mappings from one execution state to the next (based on model structure and input). This is called an *operational semantics*. In the case of our statechart language, we would need to add some sort of active state indicator to the domain. We could then define a transformation from the language domain back to itself which alters the active state based on some input event (e.g. `power_release` as seen in the diagram in Figure 4a) and the structure of the model. To completely specify the operational semantics of this language (such that it is executable), we would also need to specify the semantics of the action/condition language.

Another way to specify the semantics of a modeling language is by mapping it to another language. In general, this is known as *translational semantics*. Of course, the usefulness of such a specification depends on the target language of the mapping, which we call the *semantic domain*. One option is to use a formal mathematical notation as the semantic domain, in which case we usually call the semantics a *denotational semantics*. Another option is to combine this approach with operational semantics by specifying a mapping from the language domain to one that has a well-defined operational semantics. This is essentially what a programming language compiler does, translating from some programming language to a machine language for which the operational semantics is realized by the hardware.

We have found that translational semantics is effective for specifying the meaning of modeling languages for which an operational specification is either inconvenient or infeasible. In the case of CPS, we integrate models which describe physical processes alongside those which describe computational processes. The behavior of these physical processes is described by continuous trajectories, and so we specify the semantics of these languages via a mapping to a system of differential equations. More details

on this topic can be found in [SLL+13a].

The techniques described in this paper are based on translational semantics. For our statecharts language, we have used the specification approach involving a combination of translational and operational semantics, although the operational semantics is not specified in FORMULA. This is not a problem since our discussion in this paper is only concerned with the translational aspect of the semantics.

An operational semantics for a subset of the Stateflow language, a common variant of statecharts, is defined in [HR07]. The authors develop a mathematical notation, shown in Figure 5a, corresponding to Stateflow's standard visual representation and define execution step mappings on this notation. Their specification is sufficient to define the behavior of our language. To use their notation as a semantic domain, we have created a FORMULA domain matching its structure, which can be seen in Figure 5b. In some cases, our encoding adds subscripts to differentiate elements. (e.g. The source notation denotes (a, a, a) a triple of actions within a state definition that gives the enter action, during action, and exit action respectively. We distinguish between them as `a_en`, `a_du`, and `a_ex`.) This notation is essentially at the same level of abstraction as our language, but has significant notational differences, some of which include:

1. In our language, transitions are top-level elements which have a source and destination attribute whereas in this notation they are owned by the source and have a destination attribute.

2. The source and destination attributes of transitions in our language refer directly to the relevant state or junction whereas in this notation the destination either directly refers to a junction or gives a list, called a path, giving the containment hierarchy of the relevant state in top-down sequence.

3. Our language uses a set of relation terms to describe concepts such as the set of elements contained within another element whereas this notation uses lists (e.g. as an attribute on the container).

The source paper [HR07] should be consulted for further details.

Unfortunately, the mapping to this domain, which we call the *semantic mapping*, is too large to be included in its entirety, partly because the semantic domain uses a list-based notation which requires considerable work to construct in the FORMULA transformation. On the other hand, the notational disparities between the two language domains makes for a good example as there are many opportunities for errors. Some of the details of the transformation, however, are described as needed in Sections 6 and 7.

6 Validation Axioms

The semantic mapping is typically considered an axiom/definition – the reference point for which implementation behavior is compared to determine correctness. However, it may be that it does not properly implement, or contradicts, the informal interpretation held by the language designer. We emphasize that the formal framework may be flawless: precise, unambiguous, even formal. But if it contradicts the intentions of the language designer, the result is a counterintuitive DSML. This situation imposes real validation requirements on the semantic mapping itself.

$$
\begin{array}{lrcl}
\text{composition} & C & = & \texttt{Or}(s_a, p, T, SD) \mid \texttt{And}(b, SD) \\
\\
\text{state definition} & sd & = & ((a, a, a), C, T_i, T_o, J) \\
\text{state definition list} & SD & = & \{s_0 : sd_0; ...; s_n : sd_n\} \\
\\
\text{junction definition list} & J & = & \{j_0 : T_0; ...; j_n : T_n\} \\
\\
\text{transition} & t & = & (e_t, c, a, a, d) \\
\text{transition list} & T & = & \emptyset_T \mid t.T
\end{array}
$$

state	s			active state	s_a	$=$	$\emptyset_s \mid s$
junction	j						
path	p	$=$	$\emptyset_p \mid s.p$	destination	d	$=$	$p \mid j$
event	e			transition event	e_t	$=$	$\emptyset_e \mid e$
action	a			condition	c		

(a) Mathematical notation as defined by Hamon in [HR07]

```
1  domain Stateflow
2  {
3    NullType ::= {NULL}.
4
5    Composition ::= OrComp + AndComp.
6    primitive OrComp ::= (s_a:ActiveState, p:Path, T:TransitionList,
7      SD:StateDefList).
8    primitive AndComp ::= (active:Boolean, SD:StateDefList).
9
10   primitive StateDef ::= (a_en:String, a_du:String, a_ex:String, C:Composition,
11     T_i:TransitionList, T_o:TransitionList, J:JuncDefList).
12   StateDefList ::= StateDefListNode + NullType.
13   primitive StateDefListNode ::= (s:String, sd:StateDef, rest:StateDefList).
14
15   JuncDefList ::= JuncDefListNode + NullType.
16   primitive JuncDefListNode ::= (j:String, T:TransitionList, rest:JuncDefList).
17
18   primitive Transition ::= (e_t:TransitionEvent, c:String, a_c:String,
19     a_t:String, d:Destination).
20   TransitionList ::= TransitionListNode + NullType.
21   primitive TransitionListNode ::= (t:Transition, rest:TransitionList).
22
23   Path ::= NullType + PathNode.
24   primitive PathNode ::= (s:String, rest:Path).
25
26   ActiveState ::= String + NullType.
27   Destination ::= Path + String.
28   TransitionEvent ::= String + NullType.
29 }
```

(b) FORMULA domain describing notation structure

Figure 5 – Stateflow language used as semantic domain

Axiom Description	Inconsistency Query
Each `State` element from the input model must be mapped to an output model element of the appropriate type. (e.g.: `State` → `Path` is not allowed.)	WrongStateMapping
Each `Transition` from the input model should only be mapped to an output model element of the appropriate type. (e.g.: `Transition` → `StateDef` is not allowed.)	WrongTransitionMapping
Each `Transition` from the input model must map to a `Transition` in the output model.	MissingTransitionMapping
No `Transition` from the input model should be mapped to more than one `Transition` in the output model.	TooManyTransitionMappings
Each `State` from the input model should be mapped to both a `StateDef` and a `Composition`.	MissingStateMapping
No `State` from the input model should be mapped to more than one `StateDef` or `Composition`.	TooManyStateMappings
The length of a `Path` to a state must match the depth of the state in the input model.	WrongStatePathLength

Table 1 – Validation axioms

One solution that FORMULA provides is the ability to execute the transformation on test cases. This, while useful in itself, provides no formal or precise mechanism to conclude that the transformation is consistent with the language designer's assumptions for the semantic mapping. Of course, in order to achieve this, we first need to identify the designer's informal assumptions and formalize them as a set of axioms. Table 1 gives an overview of some of these axioms we have identified for the semantic mapping of our statecharts language.

We note that we are not presenting a technique for deriving a meaningful and complete set of validation axioms. This is left to the intuition of the language designer, although it can have other sources as well (e.g. the language requirements). We have not changed the language designer's responsibility to create a meaningful specification. What we present is an approach for mitigating the abstraction level gap between a detailed semantic specification and its author's intentions and validating that this gap does not introduce inconsistencies.

If the semantic mapping were specified in QVT [OMG08], as is common for visual modeling language transformations, we could encode these axioms as OCL constraints and check that they are preserved for specific input models. We use FORMULA instead as it supports a higher level validation approach by searching for instances which violate the constraints. This is described in Section 7.

We formalize these validation axioms within FORMULA as queries which identify the conditions under which they are violated and insert them into the semantic mapping transformation. For example, the following query identifies situations in which a transition from the input model is mapped to an incorrect element type in the semantic domain:

```
WrongTransitionMappingTypes ::= out1.NullType + out1.StateDef + out1.Composition
  + out1.StateDefList + out1.JuncDefList + out1.TransitionList + out1.Path.
WrongTransitionMapping :=
```

```
MappingRelation(trans, target), trans in in1.Transition,
target in WrongTransitionMappingTypes.
```

We specify all of the invalid types in a union type signature and query over instances of the `MappingRelation` term which relate an `in1.Transition` term to any term belonging to this union type. The identifiers `in1` and `out1` refer to the input and output namespaces within the transformation. The `MappingRelation` term is generated by each rule in the transformation which generates output model terms. This is accomplished manually with rules that take the following form:

```
MappingRelation(in_trans, out1.Transition(a,b,c,d,e)),
out1.Transition(a,b,c,d,e) :-
  ...
```

In this way, for each matching of the right-hand side of the rule we generate a transition in the output model as well as a temporary term which records the input model term that we mapped to it. Summarizing, we identify cases where a transition from the input model is mapped to something it should not be in the output model.

This approach of instrumenting the transformation with inconsistency queries could be compared to assertions about the run-time behavior of a program inserted into the text of the program. It is distinguished significantly in two ways: (i) FORMULA's declarative nature means that the location of the queries within the transformation is not important and (ii) assertions are typically checked at run-time to validate that a program behaves correctly on a given input configuration, whereas we use these queries as a search goal for synthesizing input configurations for which the transformation does not behave correctly. Section 7 contains examples of this validation process.

7 Resolving Errors

We claim that a semantic specification can have errors, in particular with respect to agreement with the informal assumptions made by the language designer. We have shown how we formalized the designer's assumptions in order to pose the issue as a validation problem. We will now present some examples illustrating synthesized counterexamples and the resolution of inconsistencies between the semantic mapping and the formalized assumptions about it.

7.1 Semantic Mapping Errors

In FORMULA, we can define partial models, which are simply models that leave some details unspecified. Figure 6 contains an example of a partial model in our statecharts language. The _ character represents an unspecified subterm within a term. This partial model contains three states with unspecified decomposition, ordering index, and containment hierarchy as well as two transitions with unspecified source, destination, and ordering index. It is not interesting from a behavioral perspective, but we are concerned primarily with structural properties of the semantic mapping, which are realized by the validation axioms. The key idea is that this partial model specifies a search space which may contain instances under which the structural properties in question do not hold.

The operational semantics specified for the semantic domain in [HR07] leaves the condition/action language abstract. As such, the particular values of the actions and conditions for states and transitions are of little importance to this exercise. We

```
1 partial model m2 of StatechartLanguage {
2   State("s1", "s1_enter", "s1_during", "s1_exit", _, _)
3   State("s2", "s2_enter", "s2_during", "s2_exit", _, _)
4   State("s3", "s3_enter", "s3_during", "s3_exit", _, _)
5   StateToTransConnectorContainment(_, _)
6   StateToTransConnectorContainment(_, _)
7   Transition("t1", _, _, "t1_trig", "t1_cond", "t1_cond_action", "t1_action", _)
8   Transition("t2", _, _, "t2_trig", "t2_cond", "t2_cond_action", "t2_action", _)
9 }
```

Figure 6 – Partial model of statecharts language domain

have used simple identifiers in this example such as `s1_enter` to indicate the enter action of state `s1`, in order to easily discern what these items are mapped to by the transformation.

We can use FORMULA to try to find a completion for this partial model under specific search goals, such as conformance to its domain. We can also supply an expression indicating the application of the semantic mapping transform to the partial model as a search term and use the transform's validation axiom queries (Table 1) as search goals. Since the queries identify the conditions under which the axioms do not hold, we effectively synthesize a model which proves the inconsistency between the language designer's intentions and the behavior of the semantic mapping.

To illustrate the use of this model finding approach to identify and correct errors in the semantic mapping, we can search for a completion of the partial model which will satisfy the conformance requirements of its domain as well as violate the validation axiom that each state's path in the semantic mapping should have a length equal to the state's containment depth in the input model by using the following query as a goal:

```
Goal := WrongStatePathLength & in1.conforms.
```

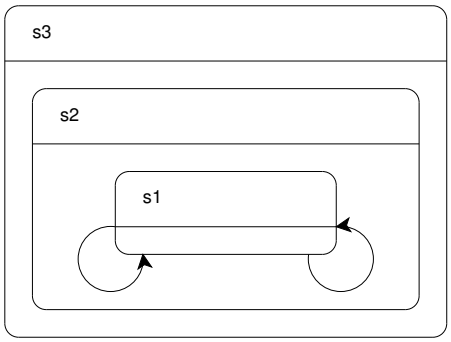
Figure 7a shows a completion for this search problem. The key feature of this completion is that we have a state (s1) contained within a state (s2) contained within a state (s3), as can be seen in its diagram in Figure 7b. The semantic mapping achieved by applying our transformation to this completion can be found in Figure 7c. We have added aliases to the FORMULA models where possible to make them more compact and easier to read, and the abstract actions and conditions are not shown in the diagram as they add unnecessary clutter.

The problem with the semantic mapping of this model is that the `PathNode` list to each state is of length 1 (second subterm of each `OrComp` term), despite the varying containment depths of the states. Based on the validation axioms, we would expect state `s2` to have a path length of 2 since it is contained within state `s1` and we would expect state `s1` to have a path length of 3 since it is further contained within state `s2`. We consider this an error in the semantic mapping.

The root cause of this error was with the way the FORMULA transformation computed a `PathNode` list to a state. In the semantic domain, this list gives a path to a state starting at a top-level state and traversing down the containment hierarchy, which is used in the operational semantics in [HR07] to easily determine the lowest common ancestor between transitioned states. The transformation first computes these paths in a bottom-up direction and then reverses them for each state. The rule which reverses the paths maps individual nodes in the bottom-up list to their corresponding nodes in the top-down list, and thus the first node in the former maps

```
model m2comp of StatechartLanguage {
  s1 is State("s1","s1_enter","s1_during","s1_exit",OR_STATE,1)
  s2 is State("s2","s2_enter","s2_during","s2_exit",OR_STATE,1)
  s3 is State("s3","s3_enter","s3_during","s3_exit",OR_STATE,1)
  StateToTransConnectorContainment(s2,s1)
  StateToTransConnectorContainment(s3,s2)
  Transition("t1",s1,s1,"t1_trig","t1_guard","t1_action_cond","t1_action",2)
  Transition("t2",s1,s1,"t2_trig","t2_guard","t2_action_cond","t2_action",1)
}
```

(a) Completion of partial model



(b) Visualization of model completion

```
model m2sem of Stateflow {
  s1comp is OrComp(NULL,PathNode("s1",NULL),NULL,NULL)
  s2comp is OrComp(NULL,PathNode("s2",NULL),NULL,StateDefListNode("s1",s1sd,NULL))
  s3comp is OrComp(NULL,PathNode("s3",NULL),NULL,StateDefListNode("s2",s2sd,NULL))
  s1sd is StateDef("s1_enter","s1_during","s1_exit",s1comp,NULL,
    TransitionListNode(t2,TransitionListNode(t1,NULL)),NULL)
  s2sd is StateDef("s2_enter","s2_during","s2_exit",s2comp,NULL,NULL,NULL)
  s3sd is StateDef("s3_enter","s3_during","s3_exit",s3comp,NULL,NULL,NULL)
  StateDefListNode("s3",s3sd,NULL)
  t1 is Transition("t1_trig","t1_guard","t1_action_cond","t1_action",
    PathNode("s1",NULL))
  t2 is Transition("t2_trig","t2_guard","t2_action_cond","t2_action",
    PathNode("s1",NULL))
}
```

(c) Semantic mapping

Figure 7 – Error-causing example

to the last node in the latter, which is a one-element list. We resolved this error by taking the mapping of the last node in the bottom-up list instead of the first.

This illustrates the concept that when a semantic specification is expressed on a low level of abstraction, it can start to resemble an implementation and exhibit defects in much the same way. By this we mean that the specification disagrees with its author's understanding of it. This is why we promote the use of a redundant, higher-level specification which corresponds more directly to the intentions of the language designer.

7.2 Structural Semantics Errors

Our validation efforts in this paper are primarily focused on the semantic mapping transformation, but this approach often uncovers issues in the structural semantics specification of the language. We demonstrate this with an example that is somewhat at the discretion of the language designer, but which we choose to resolve in the structural semantics.

We use the same partial model as before from Figure 6 and search for a completion which satisfies the conformance of of the language domain and results in a semantic mapping which has more state definitions than the input model. This is accomplished using the following query as a goal:

```
Goal := TooManyStateMappings & in1.conforms.
```

The completion we found can be seen in Figure 8a with its diagram in Figure 8b. Notably, there are two states, s2 and s3, contained within a parent state s1, and the order indices on states s2 and s3 are not adjacent. We can see in its semantic mapping, shown in Figure 8c, that the state `s1` is mapped to two different state definitions: one which contains state `s2` and one which contains state `s3`. Additionally, two top-level state lists are created.

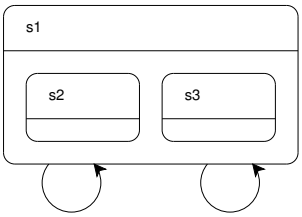
The reason for this is that the transformation created separate substate lists for each of `s2` and `s3`. In the semantic domain, the execution order of parallel states and multiple active transitions is determined by the order of items in lists. As such the transformation bases the ordering of the lists on the `Order` attribute of states (or transitions). However, the transformation is written such that it assumes that these values will be adjacent among items that go into the same list. The result is that when they are not adjacent, a list is created for each set of adjacent values. We create temporary terms in the transformation that pair these lists with the element they belong to. These temporary terms appear on the right-hand side of rules that generate output model elements, resulting in multiple output model elements being generated (since the right-hand side is matched for each additional list pairing).

This can be considered a problem in either the input domain's structural semantics specification or in the semantic mapping transformation. More accurately, the problem is a disagreement between the two which could be resolved by a change to either. We take the position that it is better in this case to alter the structural semantics for multiple reasons, namely: (i) it is easier in FORMULA to build a list ordered by sequential values than it is to do so from arbitrary values which must be sorted, and (ii) the necessary changes for the structural semantics constitute adding constraints, which will reduce the language domain; on the other hand, changing the transformation would require that any other validation exercises be repeated.

Our solution is to extend the structural semantics with the requirement that within particular contexts (e.g. states contained by the same parent state, transitions from

```
model m2comp2 of StatechartLanguage {
  s1 is State("s1","s1_enter","s1_during","s1_exit",OR_STATE,1)
  s2 is State("s2","s2_enter","s2_during","s2_exit",OR_STATE,1)
  s3 is State("s3","s3_enter","s3_during","s3_exit",OR_STATE,3)
  StateToTransConnectorContainment(s1,s2)
  StateToTransConnectorContainment(s1,s3)
  Transition("t1",s1,s1,"t1_trig","t1_guard","t1_action_cond","t1_action",2)
  Transition("t2",s1,s1,"t2_trig","t2_guard","t2_action_cond","t2_action",1)
}
```

(a) Completion of partial model



(b) Visualization of model completion

```
model m2comp2sem of Stateflow {
  s1compa is OrComp(NULL,PathNode("s1",NULL),NULL,
    StateDefListNode("s2",s2sd,NULL))
  s1compb is OrComp(NULL,PathNode("s1",NULL),NULL,
    StateDefListNode("s3",s3sd,NULL))
  s2comp is OrComp(NULL,PathNode("s1",PathNode("s2",NULL)),NULL,NULL)
  s3comp is OrComp(NULL,PathNode("s1",PathNode("s3",NULL)),NULL,NULL)
  s1sda is StateDef("s1_enter","s1_during","s1_exit",s1compa,NULL,
    TransitionListNode(t2,TransitionListNode(t1,NULL)),NULL)
  s1sdb is StateDef("s1_enter","s1_during","s1_exit",s1compb,NULL,
    TransitionListNode(t2,TransitionListNode(t1,NULL)),NULL)
  StateDefListNode("s1",s1sda,NULL)
  StateDefListNode("s1",s1sdb,NULL)
  t1 is Transition("t1_trig","t1_guard","t1_action_cond","t1_action",
    PathNode("s1",NULL))
  t2 is Transition("t2_trig","t2_guard","t2_action_cond","t2_action",
    PathNode("s2",NULL))
}
```

(c) Semantic mapping

Figure 8 – Example exhibiting structural semantics issue

the same source), the order value must be an increasing sequence of adjacent integers starting at 1. We add this requirement to the structural semantics by creating queries which identify models that violate it and then adding the logical negation of these queries to the language domain's `conforms` query. The following query applies to the particular context of states contained by the same parent state:

```
InvalidSubstateOrder :=
  in1.StateToTransConnectorContainment(parent,in1.State(_,_,_,_,_,n)), n!=1,
  m=n-1, no in1.StateToTransConnectorContainment(parent,in1.State(_,_,_,_,_,m)).

conforms :=
  !InvalidSubstateOrder & ...
```

We note that there are other constraints for the language domain which could be discovered using this approach. For example, states (and other `TransConnector` types) should not be contained within more than one state. We could discover this particular problem with the same goal query used above, since states contained within more than one parent would result in multiple paths to each state, which would also result in mappings to duplicate state definitions in the semantic domain. This is clearly a structural semantics issue since it does not make sense for states to be contained within more than one parent.

8 Critical Discussion

The key insight we propose is that a semantic mapping for a modeling language can be validated against the informal intentions of its designer. We demonstrate that these intentions can be formalized as a set of axioms over the semantic mapping in order to facilitate reasoning about the consistency between the two. Validating (or verifying) this consistency increases confidence in the semantic mapping in two respects:

1. The validation axioms constitute a redundant semantic specification for the modeling language in axiomatic form. Inconsistency between redundant specifications implies an error in at least one of them.

2. The language designer's intentions for the semantic mapping can become obfuscated by the details necessary to precisely and completely realize it. The validation axioms are less detailed and thus it is easier for the designer to judge the correspondence of them to his intentions.

We emphasize that this consistency does not imply correctness of the semantic mapping (which is in principle a tautology), but rather that inconsistency implies and helps to identify mistakes.

We present a framework prototype, based on FORMULA, which supports the proposed validation approach. This framework allows for partial automation of the workflow. The modeling language domain specification in FORMULA is automatically generated from a metamodel. We specify the semantic domain and the semantic mapping manually as we do not have counterparts for them in visual modeling tools. The framework could be extended to translate these artifacts as well, if desired. Validation axioms are specified as queries in the semantic mapping. Finally, we specify partial models manually and perform automated goal-based completion on them. This exposes the analyst to some of the features of SMT solvers without requiring expertise in them. Expertise with FORMULA is required, but we have found it to

be a relatively easy language to learn, especially given prior experience with logic programming.

Despite its advantages, we have encountered challenges within our FORMULA-based framework. In general, the automated formal validation or verification of a model transformation is a hard problem. In particular, the partial model completion operation scales poorly based on the size of the language domain specification, the semantic mapping, and the partial model. A detailed exploration of this scalability is beyond the scope of this work. There are also techniques which can be used to mitigate this issue, an investigation of which we leave as a topic for further research. These could include decomposing the semantic mapping based on independent components, using many small, focused partial models as opposed to a few large ones, and creating custom search algorithms for the partial models. We emphasize, however, that our semantic mapping validation approach is independent of the particular formalization framework and analysis workflow, the choice of which should be based on the needs of individual projects.

9 Related Work

There are numerous papers that deal with semantic mappings. For this paper we summarized a few that help to clarify the main concepts in Section 2. In addition to those, [HKR+07] adds algebraic operators for model compositions.

There are several environments that support automated semantic transformations besides FORMULA, such as Maude [RGdLV09] and Alloy [ABK07]. FORMULA differs from these tools by featuring integration with an SMT-solver and a logic-programming paradigm over algebraic data types.

This paper includes a description of our approach to specifying modeling languages and their semantics within FORMULA. We have published a number of papers previously which deal with this topic from different perspectives. We consider the advantages of parallel operational and denotational semantic specifications in [LSM+13]. We present a framework for the specification of DSMLs based on Cyber-Physical Systems in [SLL+13a] and in [SLL+13b] we describe the specification of formal semantics for Cyber-Physical components. In [SLN+12] we discuss our structured approach for specifying the semantics of large, heterogeneous, and continuously evolving modeling languages, which we have termed semantic backplane.

The approach presented in this paper is comparable to one found in [CCGT09] by Combemale et al. in that both address the issue of affirming a semantic mapping. The cited paper deals with verifying the equivalence between a reference semantics and a secondary semantics. The idea is that the reference semantics is an operational semantics at the abstraction level of the language and the secondary semantics is specified via a mapping to another domain with operational semantics. The mapping is verified with a bisimulation proof. We consider this approach to be more effective in the case where the domain and range of the semantic mapping have operational specifications, but it is not applicable if either does not. (They could be added, but we consider this infeasible in some cases.) Our approach, by contrast, constitutes a validation of the semantic mapping against the informal intentions of the language designer. We accomplish this by creating a formalized set of axioms which reflect these intentions and reasoning about the consistency of the semantic mapping with these axioms. As such, our approach is applicable in cases where there is no method for defining the equivalence of language models and semantic models other than the

semantic mapping itself.

There are tools and methods for automated and semi-automated verification of model transformations [Sch10, ALL10, GdLW+13, VP03a] as categorized in [ALS+12]. Schätz discusses the verification of declarative relational model transformations using an interactive theorem prover in [Sch10]. Our work differs primarily by considering the special case where the transformation is a semantic mapping and by the use of informal design intentions as a validation basis.

Guerra et al. propose a declarative language in [GdLW+13] for expressing "transformation contracts" in a way that is independent of the language used by the transformation. These contracts are compiled to QVT in order to verify them on a set of models. Again, our work differs by considering the special case of a semantic mapping. Futhermore, we strive for a higher level of validation by searching for models which demonstrate inconsistency.

An automated approach for verifying that model transformations preserve properties is presented by Varró and Pataricza [VP03a] which consists of verifying a property P on an input model using a model checker, transforming the model, and validating a corresponding property Q on the output model, also using a model checker. (The intent is that Q is equivalent to P, but in the output modeling domain.) The paper places the focus on determining whether a transformation preserves certain properties in particular transformed models, which differs from our focus of validating correspondence properties of the transformation in general.

A recent overview of the approaches to model transformation verification can be found in [CS13]. We are not aware of any other contributions that provide a discussion of formal semantics, emphasize the translation of informal design intentions and intuitions into formal validation axioms, and provide methods for validating the consistency of the specification.

10 Conclusion

In the DARPA Adaptive Vehicle Make (AVM) program, we have addressed the challenge of specifying the semantics of a Cyber-Physical Systems integration language [SLN+12]. The method described in this paper was developed in this context, where we must maintain a structured semantic specification of a large, heterogeneous, and continuously evolving modeling language. The size and complexity of this specification requires that we validate that it behaves as expected.

In this paper we have detailed an approach for validating that the semantics of a modeling language is consistent with the designer's intentions (or potentially with the behavior of already-existing tools). To this end, we advocate formalizing a set of axioms that directly reflect the designer's informal intentions about the semantic mapping. Our case study outlines a framework for validating the consistency of these axioms with the semantic mapping. Alternately, this consistency could be established formally as a verification. This would still constitute a validation of the semantic mapping since we cannot in principle know that the validation axioms are correct any more than we can know that the semantic mapping is correct.

We have provided a formalization framework based on FORMULA and presented a case study for our validation approach within this framework. The case study regards a statecharts language which we specify the semantics of via a mapping to Stateflow as defined in [HR07]. We demonstrate with examples that it is possible for a semantic mapping to behave in unintended ways and that our approach can be

instrumental in identifying this unintended behavior.

References

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[ALL10] Márk Asztalos, László Lengyel, and Tihamér Levendovszky. Towards automated, formal verification of model transformations. In *Software testing, verification and validation (icst), 2010 third international conference on*, pages 15–24. IEEE, 2010. doi:10.1109/ICST.2010.42.

[ALS+12] Moussa Amrani, Levi Lucio, Gehan Selim, Benoıt Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R Cordy. A tridimensional approach for studying the formal verification of model transformations. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 921–928, 2012. doi:10.1109/ICST.2012.197.

[CCGT09] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on semantics definition in mde-an instrumented approach for model verification. *Journal of Software*, 4(9):943–958, 2009. doi:10.4304/jsw.4.9.943-958.

[Che76] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. doi:10.1145/320434.320440.

[CS13] Daniel Calegari and Nora Szasz. Verification of model transformations. *Electron. Notes Theor. Comput. Sci.*, 292:5–25, March 2013. URL: http://dx.doi.org/10.1016/j.entcs.2013.02.002, doi:10.1016/j.entcs.2013.02.002.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer Heidelberg, 2006. doi:10.1007/3-540-31188-2.

[GdLW+13] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1):5–46, 2013. doi:10.1007/s10515-012-0102-y.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987. doi:10.1016/0167-6423(87)90035-9.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An algebraic view on the semantics of model composition. In *Model Driven Architecture-Foundations*

and Applications, pages 99–113. Springer, 2007. doi:10.1007/978-3-540-72901-3_8.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of s̈emantics?̈ *Computer*, 37(10):64–72, 2004. doi:10.1109/MC.2004.172.

[HR07] Gregoire Hamon and John Rushby. An operational semantics for stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5-6):447–456, 2007. URL: http://dx.doi.org/10.1007/s10009-007-0049-7, doi:10.1007/s10009-007-0049-7.

[JS09] Ethan Jackson and Janos Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software & Systems Modeling*, 8(4):451–478, 2009. doi:10.1007/s10270-008-0105-0.

[JSB12] Ethan K Jackson, Wolfram Schulte, and Nikolaj Bjørner. Detecting specification errors in declarative languages with constraints. In *Model Driven Engineering Languages and Systems*, pages 399–414. Springer, 2012. doi:10.1007/978-3-642-33666-9_26.

[KS08] Gabor Karsai and Janos Sztipanovits. Model-integrated development of cyber-physical systems. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 46–54. Springer, 2008. doi:10.1007/978-3-540-87785-1_5.

[LBM+01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001. doi:10.1109/2.963443.

[LNS+12] Zsolt Lattmann, Adam Nagel, Jason Scott, Kevin Smyth, Joseph Porter, Sandeep Neema, Ted Bapty, Janos Sztipanovits, Johanna Ceisel, Dimitri Mavris, et al. Towards automated evaluation of vehicle dynamics in system-level designs. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1131–1141. American Society of Mechanical Engineers, 2012. doi:10.1115/DETC2012-71378.

[LSM+13] David Lindecker, Gabor Simko, Istvan Madari, Tihamer Levendovszky, and Janos Sztipanovits. Multi-way semantic specification of domain-specific modeling languages. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 20–29. IEEE, 2013. doi:10.1109/ECBS.2013.29.

[OMG03] UML OMG. 2.0 ocl specification. *Object Management Group*, 2003.

[OMG08] QVT OMG. Meta object facility (mof) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[PP13] Beatriz Pérez and Ivan Porres. Reasoning about uml/ocl models using constraint logic programming and mda. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 228–233, 2013.

[RGdLV09] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling

languages with maude. In *Software Language Engineering*, pages 54–73. Springer, 2009. doi:10.1007/978-3-642-00434-6_5.

[SB81] HP Sankappanavar and Stanley Burris. A course in universal algebra. *Graduate Texts Math*, 78, 1981.

[Sch10] Bernhard Schätz. Verification of model transformations. *Electronic Communications of the EASST*, 29, 2010.

[SLL+13a] Gabor Simko, David Lindecker, Tihamer Levendovszky, Ethan K Jackson, Sandeep Neema, and Janos Sztipanovits. A framework for unambiguous and extensible specification of dsmls for cyber-physical systems. In *IEEE 20th International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, pages 30–39, 2013. doi:10.1109/ECBS.2013.30.

[SLL+13b] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of cyber-physical components with formal semantics – integration and composition. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2013. doi:10.1007/978-3-642-41533-3_29.

[SLN+12] Gabor Simko, Tihamer Levendovszky, Sandeep Neema, Ethan Jackson, Ted Bapty, Joseph Porter, and Janos Sztipanovits. Foundation for model integration: Semantic backplane. In *Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE*, pages 12–15, 2012. doi:10.1115/DETC2012-70534.

[VP03a] Dániel Varró and András Pataricza. Automated formal verification of model transformations. *CSDUML*, pages 63–78, 2003.

[VP03b] Dániel Varró and András Pataricza. Vpm: Mathematics of metamodeling is metamodeling mathematics. *Journal of Software and Systems Modelling*, 1:1–24, 2003.

[WNO+12] Ryan Wrenn, Adam Nagel, Robert Owens, Di Yao, Himanshu Neema, Feng Shi, Kevin Smyth, Joseph Porter, Ted Bapty, Sandeep Neema, et al. Towards automated exploration and assembly of vehicle design models. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1143–1152. American Society of Mechanical Engineers, 2012. doi:10.1115/DETC2012-71464.

About the authors

David Lindecker is a Graduate Research Assistant for the Institute for Software Integrated Systems at Vanderbilt University. His interests include programming languages, verification, and algorithms. Contact him at david.a.lindecker@vanderbilt.edu.

Gabor Simko is a Software Engineer at Google Inc. He received a PhD in computer science from Vanderbilt University. His interests include formal specification and verification of modeling languages. Contact him at `gabor.simko@vanderbilt.edu`

Tihamer Levendovszky is a Research Assistant Professor at Vanderbilt University. He received his PhD from the Budapest University of Technology and Economics. His interests include model-based engineering and performance analysis of software systems. Contact him at `tihamer.levendovszky@vanderbilt.edu`.

Istvan Madari is a Staff Engineer at Vanderbilt University. His interests include model-based integration, verification, and validation of Cyber-Physical Systems. Contact him at `istvan.madari@vanderbilt.edu`.

Janos Sztipanovits is an E. Bronson Ingram distinguished professor in the Department of Electrical and Computer Engineering and director of the Institute for Software Integrated Systems at Vanderbilt University. Sztipanovits received a PhD in electrical engineering from the Technical University, Budapest. Contact him at `janos.sztipanovits@vanderbilt.edu`.

Acknowledgments This research is supported by the AVM Program of the Defense Advanced Research Project Agency (DARPA) under award # HR0011-12-C-0008 and the National Science Foundation under award # CNS-1035655.