Lost in Translation: Forgetful Semantic Anchoring

Daniel Balasubramanian
Vanderbilt University
Institute for Software Integrated Systems
Nashville, TN
daniel@isis.vanderbilt.edu

Ethan K. Jackson
Microsoft Research
Redmond, WA
ejackson@microsoft.com

***Abstract*—Assigning behavioral semantics to domain-specific languages (DSLs) opens the door for the application of formal methods, yet is largely an unresolved problem. Previously proposed solutions include *semantic anchoring*, in which a transformation from the DSL to an external framework that can supply both behavioral semantics and apply formal methods is constructed. The drawback of this approach is that it loses the structural constraints of the original DSL along with the details of the transformation, which can lead to erroneous results when formal methods are applied. We demonstrate this problem of "forgetful" semantic anchoring using existing approaches through a translation from dataflow systems to interface automata. We then describe our modeling tool FORMULA and apply it to the same example, showing how forgetful semantic anchoring can be avoided.**

***Keywords*-Composition; Behavioral Semantics**

I. Introduction

Model-based development (MBD) is a methodology for engineering complex software systems through formal and composable abstractions. Many versions of MBD expose abstractions using *domain-specific modeling languages* (DSMLs) (or *domain-specific languages* (DSLs) for short). In this case, a *model* is a description expressed through the abstract syntax of one or more DSLs. This approach requires each DSL to provide (1) a *structural semantics* for representing models and rejecting erroneous instances of DSL syntax, (2) a *behavioral semantics* for relating models to *traces*, *runs*, or other descriptions of dynamics. A major benefit of this approach is the ability to define abstractions that appear drastically different from each other, while reusing a common set of behavioral semantics and formal methods. For example, *labeled-transition systems* (LTSs), *interface automata*, and *petri nets* are all well-studied low-level abstractions with rich formal methods. These frameworks are commonly used to assign behavioral semantics to DSLs, and their formal methods are used to deduce properties of models.

The DSL approach does require effort to reuse behavioral semantics. The most common technique, called *semantic anchoring*, employs a *model transformation* τ to translate models from DSL D_X *without* behavioral semantics into DSL D_Y *with* behavioral semantics. The pair (τ, D_Y) then provides D_X with behavioral semantics; D_Y is called the *semantic unit*. Presumably, once a model is translated into a semantic unit, then formal methods for D_Y can be applied. However, the applicability of analysis/verification techniques relies on a crucial assumption:

The Assumption of Forgetfulness: *The formal methods of the semantic unit D_Y can be applied to models of D_X without considering the structural semantics of D_X or the details of the anchoring transformation τ.*

In other words, we can forget how a model is translated; only the result of the transformation matters. We use the term *forgetful semantic anchoring* to refer to semantic anchoring frameworks relying on this assumption.

In this paper we show that the assumption of forgetfulness does not hold in general. We provide an example where a class of dataflow systems is transformationally anchored to interface automata, but formal methods of interface automata yield erroneous results w.r.t the dataflow models. This phenomenon occurs because key constraints on the dataflow systems are *lost in translation*. Forgetting these constraints results in a decision problem which is so under-constrained that its solution is no longer useful. Our example reflects the constraints found in dataflow-based *synchronous languages*, e.g. *Lustre* and *Signal* [1], and the translations from these languages to automata-based formalisms [2]. We then discuss the reasons why semantic anchoring without the assumption of forgetfulness is non-trivial and briefly describe our modeling tool, FORMULA, and its mechanisms for performing semantic anchoring without forgetting constraints.

II. Related Work

The term *semantic anchoring* was introduced in [3] and composition of semantic units were explored in [4]. In this work, the *Model Integrated Computing* (MIC) [5] framework was used: DSL structural semantics were specified with the *MetaGME* [6] metamodeling language and model transformations were specified using the *GReaT* [7] transformation language. Semantic units, such as low-level LTS and automata formalisms, were bootstrapped with *abstract state machines* (ASMs) [8]. All DSLs were eventually reduced to ASMs. Formal methods on ASMs include explicit state enumeration, symbolic execution, and bounded symbolic model checking.

Similarly, the Atlas Model Management Architecture (AMMA) [9] also uses ASMs to define the behavioral semantics of DSLs. In this case, the OMG's *meta-object facility* (MOF) [10] serves as the metamodeling language and the *Atlas Transformation Language* (ATL) is used for model transformations. Note that ATL also reflects the OMG's QVT standard. AMMA exposes a built-in ASM DSL to make bootstrapping to ASMs simpler. These tools are built on top of the *Eclipse Modeling Framework* (EMF).

Translating between formalisms has been studied in the context of logic [11]. In this work, the component(s) of one logic, such as a proof theory, can be transferred to another logic via suitable maps that rely on limited features of the two logics involved. Concurrency theory also deals with such mappings and the properties they preserve.

In addition to ASMs, other theorem provers, including *Alloy* [12], *B* [13], and *Z* [14] have been used to evaluate properties of modeling artifacts for different purposes. However, the automatic conversion of metamodeling constraints into theorem provers has remained an open problem. (These constraints are usually expressed in OCL [15].) In this sense, much of the existing literature describes (partially) forgetful semantic anchoring approaches.

III. Case Study: SDF to Interface Automata

In this section we show an example of semantic anchoring where global structural constraints are lost in translation. The DSL to be anchored, called *synchronous dataflow* (SDF), is used to model systems with the following properties:

1) Dataflow: A system is an interconnection of dataflow operators and FIFO buffers.
2) Synchronous: The duration of computation is effectively zero with respect to the arrival rate of input events.
3) Homogeneous, Rate 1: A dataflow operator *fires* if a data token is available on every input buffer. Afterwords, a token is enqueued on every output buffer.

This is a simplification of both the dataflow-based synchronous languages [1] and the full SDF abstraction [16], which may be multi-rate and have data-dependent firing conditions. The process of translating dataflow formalisms to automata formalisms (and vice-versa) has been studied extensively [2]; we adapt them for purposes of illustration.

A. Homogeneous Synchronous Dataflow

Formally, an SDF system is a structure $DF = \langle N, P, C, in, out, d\rangle$ where:

1) N is the set of *nodes* (dataflow operators).
2) P is a set of *communication ports*.
3) $in : N \rightarrow 2^P$ is a function from nodes to sets of ports; $in(n)$ designates the input ports of node n.
4) $out : N \rightarrow 2^P$ is a function from nodes to sets of ports; $out(n)$ designates the output ports of node n.
5) $C \subseteq out(N) \times in(N)$ is a set of directed communication *channels*. A channel is directed from an output to an input.
6) $d : C \rightarrow \mathbb{N}$ is a mapping from channels to non-negative integers; $d(c)$ is called the *delay* of channel c.

The top of Figure 1 shows a small dataflow system consisting of two dataflow operators X and Y. Node X has one output port A connected to the input port E of Y. (Therefore, $in(\mathsf{X}) = out(\mathsf{Y}) = \emptyset$, $out(\mathsf{X}) = \{\mathsf{A}\}$, $in(\mathsf{Y}) = \{\mathsf{E}\}$).

The *delay* of a channel indicates the size of the underlying communication buffer. A zero-delay channel is an instantaneous data-dependency. The instant one node writes to a channel another node removes the token from the channel. Zero-delay channels cause readers to block until data becomes available. Channels with non-zero delay remember the last values d that passed through the channel. Reading nodes can read these channels at any time and writers can write them at any time with no chance of communication deadlock. We follow tradition by indicating zero-delay channels with solid lines and delay channels with dashed lines.

Dataflow systems use ports and channels to restrict communication between two sources. This requires a global constraint on port names:

$$\forall n, n' \in N, \begin{cases} in(n) \cap out(n') = \emptyset \wedge \\ in(n) \cap in(n') \neq \emptyset \Rightarrow n = n' \wedge \\ out(n) \cap out(n') \neq \emptyset \Rightarrow n = n' \end{cases} \tag{1}$$

Most synchronous languages impose the global constraint that every communication cycle must contain at least one delay edge to prevent deadlock [17].

$$\forall H \in cycles(DF) \sum_{c \in H} delay(c) > 0. \tag{2}$$

B. The Interface Automata Semantic Unit

We follow previous semantic anchoring work by assuming a translation onto an *interface automata* semantic unit defined via *abstract state machines* [18]. In the interest of space, we do not show the ASM specification. Interface automata are similar to finite-state automata, but augmented with a set of actions partitioned into three disjoint sets: input actions, internal actions, and output actions. An automaton can observe input actions, emit output actions, while internal actions occur without provocation. This partitioning of the actions forms an interface between the automaton and its environment. Interface automata are a suitable semantic unit for the SDF abstraction because SDF graphs also receive input events from their environment and create data tokens that are either passed to the environment or remain hidden within channels.

An interface automaton [19] is a structure $\mathcal{A} = \langle Q, Q_0, A_I, A_O, A_H, \rightarrow\rangle$ where:

1) Q and $\emptyset \subset Q_0 \subseteq Q$ are states and initial states, respectively.
2) A_I, A_O, A_H are mutually disjoint sets of input, output, and internal *actions*. Let $A = A_I \cup A_O \cup A_H$ denote all these actions.
3) $\rightarrow \subseteq Q \times A \times Q$ is the transition relation.

Given two interface automata $\mathcal{A}, \mathcal{A}'$, let $shared(\mathcal{A}, \mathcal{A}') = A \cap A'$. Then the *synchronous product* $\mathcal{A}'' = \mathcal{A} \otimes \mathcal{A}'$ of automata is given by:

1) $Q'' = Q \times Q',\ Q_0'' = Q_0 \times Q_0'$.
2) $A_i'' = A_i \cup A_i' - shared(\mathcal{A}, \mathcal{A}'),\ i \in \{I, O\}$.
3) $A_H'' = A_H \cup A_H' \cup shared(\mathcal{A}, \mathcal{A}')$.

Shared actions become internal actions that are no longer visible to the environment. The transition relation synchronizes on shared actions:

$$\rightarrow'' = \begin{array}{l} \left\{ ((s_0,t),\alpha,(s_1,t)) \;\middle|\; \begin{array}{l} (s_0,\alpha,s_1) \in \rightarrow \;\wedge \\ \alpha \notin shared(\mathcal{A},\mathcal{A}') \;\wedge \\ t \in Q' \end{array} \right\} \cup \\ \left\{ ((s,t_0),\alpha',(s,t_1)) \;\middle|\; \begin{array}{l} (t_0,\alpha',t_1) \in \rightarrow' \;\wedge \\ \alpha' \notin shared(\mathcal{A},\mathcal{A}') \;\wedge \\ s \in Q \end{array} \right\} \cup \\ \left\{ ((s_0,t_0),\alpha,(s_1,t_1)) \;\middle|\; \begin{array}{l} (s_0,\alpha,s_1) \in \rightarrow \;\wedge \\ (t_0,\alpha,t_1) \in \rightarrow' \end{array} \right\} \end{array} \tag{3}$$

Note that the synchronous product is defined if two automata do not share internal events, and if $A_i \cap A_i' = \emptyset$ for $i \in \{I, O\}$.

C. Transformation onto Interface Automata

To apply semantic anchoring we define a transformation from the structural semantics of SDF onto the interface automata semantic unit.

The bottom of Figure 1 shows this transformation process for the SDF graph at the top of the figure. The algorithm first converts the graph to three individual interface automata, labeled (a), (b) and (d) in the bottom of the figure: one automaton for each Node, and one automaton for each Channel with a delay of zero. Node X has no input ports and a single output port, which means that it can continually emit a token from its port named A. This is captured by the automaton labeled (a) in the bottom of the figure, which has a single state with one transition that emits an output action named A. The values of the tokens emitted by ports are not modeled, so we simply define an output action with the same name as the port. Node Y is transformed into an automaton (labeled (d) in the figure) in a similar way, except that its automaton has a single transition labeled by an input action E (corresponding to Y's input port). The channel between the ports A and E is translated into an automaton (labeled

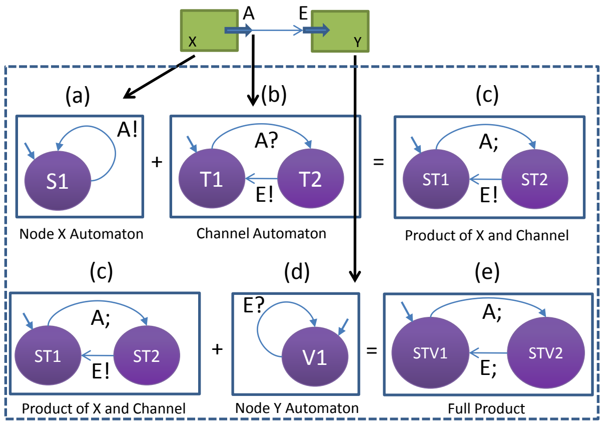

Figure 1. An SDF system with 2 nodes (top) and the resulting interface automaton (e). The automata for individual SDF components are shown in (a), (b) and (d).

(b) in the figure) with two states. The initial state, $T1$, waits for an input action named A. Upon the occurrence of this action, it takes a transition to state $T2$ and emits an output action named E, and the process repeats.

After the interface automata for the individual SDF graph components have been constructed, the second step of the transformation is the application of a *synchronous product*, which combines all automata while synchronizing on port names. We calculate the product using the definition listed above in section III-B. The bottom of Figure 1 shows the steps for constructing the product of the individual automata: we first compose Node X's automaton with the Channel's automaton, and compose that resulting automaton with Node Y's automaton, yielding the full product automaton (labeled (e) in the figure). In the first composition step, A is a shared action, and thus becomes an internal action in the composition (labeled (c)). E is a shared action in the second composition step, so that both actions are internal in the final product.

D. Applying Formal Methods

This semantic anchoring process is forgetful, as shown in Figure 2. Node Z is anchored to the interface automaton in the far left-hand side of the figure; the translation does not retain any information about the SDF model. At this stage formal methods of interface automata can be used to check *reachability*, *safety*, *deadlock-freedom*, and *liveness* properties of Z, which are local properties of the system Z. It also important to decide the global effects combining other components with Z. These properties can be more difficult to decide, because one must consider the hypothetical systems that can be connected to Z. *Compositional properties* [20] guarantee that well-behaved systems can be safely combined. These properties are key for scalability of formal methods. Unfortunately, it is precisely compositional reasoning that can be lost in translation.

From the perspective of DSLs, a compositional property is defined this way: Given a DSL D, let $models(D)$ be the set of all legal instances of DSL syntax. A property ρ is a predicate on DSL models; the property is true for model m iff $\rho(m)$ is true. A *composition operator* $\oplus$ is a partial function for composing models, i.e. $\oplus : models(D) \times models(D) \to models(D)$.

Definition III.1. Compositional Properties. Given D, $\oplus$, and a property ρ, then ρ is compositional over $\oplus$ if:

$$\forall (m, m') \in (\mathbf{dom}\ \oplus),\ \ \rho(m) \wedge \rho(m') \Rightarrow \rho(m \oplus m'). \quad (4)$$

Consider the DSLs SDF and $\mathsf{IntAutomata}$, with composition operators:

$$\oplus_{\mathsf{IntAutomata}} = \otimes, \quad \oplus_{\mathsf{SDF}} = \ _{p_o}\|_{p_i}. \quad (5)$$

where $\oplus$ for interface automata is synchronous product. The composition operators for dataflow systems are a family of *parallel product* operators where $(m\ _{p_o}\|_{p_i}\ m')$ creates the union of two dataflow systems, except that output port p_o in m is connected to p_i in m'.

Let τ be the semantic anchoring transformation from SDF to $\mathsf{IntAutomata}$ then, if $(m\ _{p_o}\|_{p_i}\ m')$ is defined, then:

$$(m\ _{p_o}\|_{p_i}\ m') = \tau(m) \otimes \tau(c_{p_i,p_o}) \otimes \tau(m'). \quad (6)$$

where $\tau(c_{p_i,p_o})$ generates the interface automata for the new channel c_{p_i,p_o} according to the previous section. Due to the global constraint of Equation 2, $_{p_o}\|_{p_i}$ is only defined for models that compose without introducing deadlock. In other words:

$$\forall (m, m') \in (\mathbf{dom}\ _{p_o}\|_{p_i})\ \rho(m) \wedge \rho(m') \Rightarrow \rho(m\ _{p_o}\|_{p_i}\ m'). \quad (7)$$

where ρ is deadlock freedom. SDF systems are compositional with respect to deadlock-freedom. Applying Equation 6,

$$\forall (m, m') \in (\mathbf{dom}\ _{p_o}\|_{p_i})$$
$$\rho(\tau(m)) \wedge \rho(\tau(m')) \Rightarrow \rho(\tau(m) \otimes \tau(c_{p_i,p_o}) \otimes \tau(m')). \quad (8)$$

Together, the structural constraints and semantic anchoring τ translate SDF systems into interface automata such that deadlock freedom is also compositional $\otimes$.

However, forgetful semantic anchoring is unable to utilize these properties, because it cannot refer either to the structural constraints of higher-level abstractions, nor to the action of the transformation. Properties can only be asked with respect to the semantic unit. To see this, we ask a hypothetical question about the danger of combining Z with other components:

$$\exists m' \in models(\mathsf{IntAutomata}),\ \neg\rho(\mathcal{A}_{\mathsf{Z}} \otimes m'), \quad (9)$$

asks if there is some interface automata that causes the dataflow component Z to deadlock ($\mathcal{A}_{\mathsf{Z}} = \tau(\mathsf{Z})$). The answer to this question is *yes*: One such interface automaton

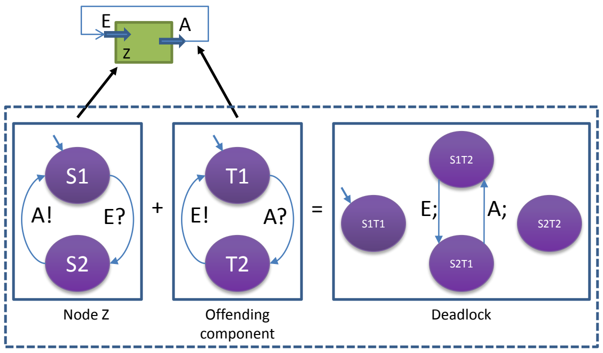

Figure 2. An example of an erroneous deadlocking component.

causing Z to deadlock is shown in the center of Figure 2. However, such an automaton corresponds to a zero-delay cycle between the input port E and the output port A. Such a composition is impossible in SDF domain. In fact, all such constructions are erroneous. This example shows that reasoning about the effects of component composition must generally take into account the rich structural semantics of higher-levels of abstractions, as well as the actions of the semantic anchoring transformation. Otherwise, semantic anchoring and compositional reasoning may not be compatible.

IV. Semantic Anchoring with FORMULA

The key idea of our approach to semantic anchoring is to specify domain constraints and model transformations using the same formalism. Our modeling language FORMULA employs an expressive fragment of logic programming to accomplish these tasks. Additionally, many formal methods can be reduced to *finite model finding* and *symbol execution* tasks over logic programs. FORMULA is coupled with a state-of-the-art satisfiability modulo theories (SMT) solver in order to provide a generic set of formal methods. Due to space constraints, we give a general outline of our procedure; for a more complete description of FORMULA, see [21].

Applying FORMULA to the example above would involve the following steps. First, define *domains* for both the SDF and Interface Automata languages. Domains are the basic unit of encapsulation in FORMULA and consist of: (1) data structures for representing models, and (2) structural constraints to check the validity of models. The structural constraints are specified using *logic-programming*. Semantically anchoring models from the SDF domain to the Interface Automata uses FORMULA's *transformation* construct. Transformations are defined using logic-programming style *rules*, i.e., *unification* procedures, to search the data contained by an input model and create elements in the output model when patterns are matched and constraints are met.

Formal methods are applied after the semantic anchoring. The key difference with our framework is that it maintains information about domain constraints and transformations, allowing formal methods to utilize the properties of high-level abstractions, even when they are anchored to low-level abstractions that may not guarantee the same properties. For example, using FORMULA to search for a non-compositional component as in Section III-D would cause our finite model finding procedure to attempt to construct two SDF models that are valid both before and after their parallel product is constructed. However, these models must also result in a deadlocking interface automaton when semantically anchored. Failure to construct a finite model for this query on the composed logic programs means that deadlock is a compositional property for this semantic anchoring.

V. Conclusions

We have shown the problems that can arise when using forgetful semantic anchoring approaches. Our example presented a domain-specific language for a dataflow abstraction that was semantically anchored onto interface automata. As a result of forgetful semantic anchoring, compositionality of deadlock freedom was lost in the translation of dataflow models to interface automata.

We then introduced our modeling framework FORMULA, which uses logic programming to consistently represent domain constraints and model transformations. By using a single framework, we described how FORMULA is able to reason about high-level properties even after semantic anchoring. Our experiments reducing the finite model finding problem to a satisfiability modulo theories solver have yielded encouraging results. Future work includes applying these techniques to larger examples to test the scalability of our approach.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan 2003.

[2] N. Halbwachs, "Synchronous Programming of Reactive Systems," in *Computer Aided Verification, 10th International Conference, (CAV)*, 1998, pp. 1–16.

[3] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specificmodeling languages," in *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, 2005, pp. 35–43.

[4] ——, "Compositional specification of behavioral semantics," in *Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2007, pp. 906–911.

[5] J. Sztipanovits and G. Karsai, "Model-integrated computing," *IEEE Computer*, vol. 30, no. 4, pp. 110–111, 1997.

[6] Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[7] A. Agrawal, G. Karsai, and Á. Lédeczi, "An end-to-end domain-driven software development framework," in *OOPSLA Companion*, 2003, pp. 8–15.

[8] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*. Oxford University Press, 1993, pp. 9–36.

[9] D. D. Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio, "Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs," April 2006, RR 06.02. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00023008/en/

[10] Object Management Group, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[11] M. Cerioli and J. Meseguer, "May i borrow your logic?" in *Mathematical Foundations of Computer Science*, 1993, pp. 342–351.

[12] D. Jackson, "A comparison of object modelling notations: Alloy, uml and z," Tech. Rep., August 1999. [Online]. Available: http://sdg.lcs.mit.edu/publications.html

[13] R. Marcano and N. Levy, "Using b formal specifications for analysis and verification of uml/ocl models," in *In Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, 2002, pp. 91–105.

[14] A. Evans, R. B. France, and E. S. Grant, "Towards formal reasoning with uml models," in *In Proceedings of the Eighth OOPSLA Workshop on Behavioral Semantics*.

[15] Object Management Group, *Object Constraint Language (OCL) Specification Version 2.0*, May 2006. [Online]. Available: http://www.omg.org/spec/OCL/2.0/

[16] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," in *Proceedings of the IEEE*, 1991, pp. 1305–1320.

[18] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic essence of AsmL," *Theor. Comput. Sci.*, vol. 343, no. 3, pp. 370–412, 2005.

[19] L. de Alfaro and T. A. Henzinger, "Interface Automata," in *Foundations of Software Engineering (FSE)*, 2001.

[20] G. Gößler, S. Graf, M. E. Majster-Cederbaum, M. Martens, and J. Sifakis, "Ensuring Properties of Interaction Systems," in *Program Analysis and Compilation*, 2006, pp. 201–224.

[21] E. K. Jackson and J. Sztipanovits, "Formalizing the structural semantics of domain-specific modeling languages," *Software and Systems Modeling*, 2008.