Formalizing the Structural Semantics of Domain-Specific Modeling Languages

ETHAN JACKSON and JANOS SZTIPANOVITS
Institute for Software Integrated Systems, Vanderbilt University

Abstract— **Model-based approaches to system design are now widespread and successful. These approaches make extensive use of model structure to describe systems using domain-specific abstractions, to specify and implement model transformations, and to analyze structural properties of models. In spite of its general importance the *structural semantics* of modeling languages are not well-understood. In this paper we develop the formal foundations for the structural semantics of domain specific modeling languages (DSML), including the mechanisms by which metamodels specify the structural semantics of DSMLs. Additionally, we show how our formalization can complement existing tools, and how it yields algorithms for the analysis of DSMLs and model transformations.**

Index Terms— **Model-based Design, Domain Specific Modeling Languages, Structural Semantics, Metamodeling, Formal Logic, Horn Logic**

I. Introduction

Domain-specific modeling languages (DSMLs) play an important role in software and system design. They are essential components of the OMG's model-driven architecture (MDA) [76], mature tools exist for constructing and utilizing DSMLs [31], [36], [64], and many methodologies in model-based design, such as platform-based design [45] and actor-based design [66] exploit the DSML metaphor [65].

Despite widespread application of DSMLs, many scientific questions remain about their formal properties [8], [41], [61], which can be loosely grouped into the *structural* and *behavioral* regimes. The structural regime concerns the specification, representation, and manipulation of models as represented in some domain-specific syntax. Research in the behavioral regime focuses on the specification and analysis of domain-specific execution semantics. Efforts to formalize *statecharts* [37], [38] and *message sequence charts* [3], [39] fit into this regime. Others have studied the general problem of linking domain-specific syntaxes with execution semantics [14], [72].

Anytime a new community adopts DSML-based modeling, its members inevitably desire a precise formalization of behaviors for the purposes of verification and simulation. This evolution can be seen in the hybrid systems [83], embedded systems [34], and security [55] communities. However, the immediacy of behavioral issues has dominated the spotlight, leaving issues in the structural regime behind.

In all fairness, it is not obvious that the foundations of DSML syntax should differ from that of existing programming languages. Traditionally, programming language construction follows a well-defined path [1]: First, language syntax is defined with an eBNF grammar [15] and a parser is generated. Second, a type-system is defined. Third, algorithms are developed that walk the abstract syntax tree (AST) and check the well-typedness of the program. The terms "domain-specific" and "model" do not by themselves indicate that the procedure should be any different for DSMLs.

The first sign that DSMLs diverge from traditional language design appears in their specification with *metamodels* [5]. Metamodels employ UML-like class diagrams to describe rich syntactic constructs with hierarchical internal structure (*aggregation*) carrying typed data (*attributes*). Metamodeling also focuses attention to relations (*associations*) between syntactic entities, providing n-ary relations over infinite sets. Unlike BNF grammars, metamodels treat these building blocks as first-class concepts. Expressive constraint languages, such as the *Object Constraint Language* (OCL) [79], enrich metamodels by supporting expressive constraints on legal model instances [55]. These observations have already led many researchers to relate metamodels to graph grammars, which natively support relations [23], [60].

The second sign of divergence occurs in the varied applications of DSMLs syntax, which include *model transformations*, *design space exploration*, and *correct-by-construction* design. Model transformations [70] translate between domain-specific syntaxes to change abstraction levels [32], compose modeling aspects [35], and relate platform-independent models with platform specific ones [76]. Modern model transformation languages utilize extended graphs grammars to capture the complexities of DSML syntax [23], [60]. Syntax has also

been used to perform design space exploration. These techniques employ syntactic perturbations to generate optimized variants of models [57], [74]. Again, the expressiveness of metamodeling constraints facilitates meaningful design space exploration. This same expressiveness can be used to project behavioral properties (e.g. deadlock freedom) onto the syntactic level [52]. Correct-by-construction design [33] uses expressive syntactic rules in conjunction with a suitable behavioral semantics to statically identify models with bad behavioral properties. Note that the correct-by-construction approach differs from *static analysis* of traditional programming languages, because DSMLs are *a priori* designed to maximize analysis. Programming language static analysis has historically evolved in the other direction: First, the language is fixed (e.g. C++), and then reasonable approaches to static analysis develop, e.g. dataflow analysis of software [28].

Given the important uses-cases for DSML syntax we might expect that:

1) There exists a precise mathematical foundation for metamodeling, model transformations, and DSML syntax.
2) Tool-independent formal descriptions of modeling artifacts can be extracted from tool-dependent artifacts.
3) The formal foundation yields analysis techniques for DSML syntax, metamodels, and model transformations.

Unfortunately, this is not the current state of affairs. *Metaprogrammable* tools (i.e. tools that use metamodels) and standardized metamodeling languages have evolved independently from each other, from formalizations of the metamodeling process, and from formalizations of model transformations. Here are some examples that illustrate this:

1) The work on *KM3* [54] provides a formal metamodeling language for use with graph transformations, but does not address expressive constraints incorporated into the source/target languages. Similar limitations are present in the *VMP* [87] formalism used by *Viatra2* [17].
2) The mature metaprogrammable Generic Modeling Environment (GME) [64] and Graph Rewriting And Transformation (GReAT) tool [29] provide expressive DSMLs and extensive model transformation features. However, the precise structural semantics of tool artifacts depend on (and are hidden in) the implementation of these complex tools.
3) Standards such as the *UML superstructure* [80], *UML infrastructure* [78], and *Meta-Object Facility* (MOF) [75] do not provide sufficiently rigid definitions of the DSML process to permit interoperability between tools. See [24] for a detailed example of this phenomenon.

In this paper we explore a *bottom-up approach* to formalizing metamodeling, DSML syntax, and model transformations. By bottom-up, we mean that our approach does not begin with the concepts of metamodeling or model transformation, but ends with these. Instead, we start with a simple formal core capable of expressing the rich syntaxes of a class $\mathbb{D}$ of DSMLs. Each member of this class is simply called a *domain* (motivated by the phrase *domain-specific*) and each domain D defines some domain-specific syntax. Next, we define a class $\mathbb{T}$ of functions that relate the syntactic elements of one domain with the elements of another; this class represents model transformations. Third, we identify a special pair $(D_{meta}, T_{meta}) \in \mathbb{D} \times \mathbb{T}$ that together are capable of generating all domains in the class; this becomes metamodeling. Finally, we define the classes $\mathbb{D}$ and $\mathbb{T}$ using the same underlying mathematical apparatus based on deductive logic.

With our approach we can formulate important properties over domains and transformations, including: Domain emptiness, domain equivalence, and structure preserving maps.

1) *Domain emptiness* refers to DSMLs with no legal syntactic instances. Empty domains are created when metamodel composition introduces inconsistencies, or when metamodeling constraints contain mistakes.
2) Two *domains are equivalent* if they have the same set of syntactic instances, even though their metamodels may differ. Domain equivalence provides a mechanism to compare domains independently from the metamodeling language.
3) *Structure preserving maps* refer to model transformations that always rewrite legal syntactic instances to legal syntactic instances.

We have developed a theorem prover called FORMULA (FORmal Modeling Using Logic Analysis) that calculates these properties when domains/transformations are described using Horn logic [53] with stratified negation [48], [51]. We will not describe the details of FORMULA here, but show some results of the tool.

Finally, we incorporated our formalization into the *Model Integrated Computing* (MIC) tool suite, which has amassed over a decade of massive modeling efforts ranging from models of the NASA space station [12] to the sensor and control networks of automotive plants [68].

We show in detail how our framework can be connected to this tried-and-tested DSML-based modeling suite. This case study illustrates that the bottom-up approach is flexible enough to adapt to various modeling tools in the face of competing standards and metamodeling languages.

This paper is divided into three main sections. Section II describes work related to formalizing DSML syntax. Section III describes our formalism in abstract terms, i.e. without a particular metamodeling language or model transformation technology in mind. We build the key concepts of *domains*, *model transformations*, and *metamodeling*. We also illustrate how this formalism yields opportunities for analysis. Section IV presents one incarnation of our formalism using *Horn logic* and links it to the MIC tools. Finally, we conclude in Section V.

II. Related Work

Our view of metamodeling differs from existing approaches based on an instance semantics [54], [87]. These approaches view metamodels as class-like constructs that can be instantiated. A model conforms to a metamodel if each modeling element is an instance of a metamodeling concept. A *meta-metamodel* is a metamodel whose conforming instances are metamodels. This top-down approach uses the meta-metamodel to drive the set of possible metamodels that drives the sets of possible models. While this approach is certainly reasonable, it makes it difficult to disassociate models from the tools and standards used to construct them. The bottom-up approach is meta-metamodel agnostic, i.e. domains and transformations exist independently from any particular metamodeling lanugage.

Formalizations and applications of expressive syntax have appeared in many different forms. Within the domain-specific language community, graph-theoretic formalisms [8], [22], [81] have received the most research attention. However, the majority of work focuses on graph rewriting systems as a foundation for model transformations. See [60], [70] for a taxonomy of existing graph-theoretic model transformation approaches. The problems of calculating properties of rich syntax, composing syntax with known properties, and constructing design space representations have not received the same attention from graph-theoretic methods. For example, the model transformation tool VIATRA [17] supports executable Horn logic (i.e. Prolog) to specify transformations, but does not focus on restricting expressiveness for the purpose of analysis.

The visibility of UML has driven researchers to formalize it semantics. This is a non-trivial task because UML includes many capabilities (diagrams) including metamodeling, state machines, activities, sequence charts (interactions), and use-case diagrams [80]. Approaches for formalizing UML must tackle the temporal nature of its various behavioral semantics, necessitating more expressive formal methods. Well-known tools/methods such as Alloy [46], B [69], and Z [26] have been used to varying degrees of success. These approaches make trade-offs between expressiveness and the degree of automated analysis. For example, Z and B proofs typically require interactive theorem provers [6], [10] and model generation may not be supported. Z or B formalizations of UML could be a vehicle for studying rich syntax, but automated analysis is less likely to be found.

Alloy, like our tool FORMULA, is less expressive than other methods, thereby supporting automated analysis [47]; it also has a recently improved model generation (model finding) procedure [86]. However, the mathematical underpinnings of Alloy are quite different from our approach: Alloy supports algebraic specifications through first-order logic with relations over atoms plus transitive closure. Contrarily, our framework is based on a non-monotonic extension of Horn logic [51]. One key difference is that FORMULA specifications can be executed like standard *logic programs* [84]. Complexity-theory also offers a coarse-grained way of comparing logic programs with other methods [19].

The BNF grammars of traditional programming languages can be extended to capture richer syntaxes. *Attribute grammars* (AGs) [82] , proposed by Knuth [59], could be the earliest example of such a mechanism. AGs allow the productions of a BNF grammar to trigger actions capable of examining tokens and attaching new data to tokens. These actions can be specified programmatically, thereby significantly increasing the power of the grammar. However, calculating properties of languages specified through AGs depends on the expressiveness of the actions. Additionally, composing AGs has proved to be a difficult task [27]. More recently, *pluggable type systems* have been studied as a mechanism to compose the type systems of traditional programming languages [4].

III. The Formal Semantics of Domains and Domain Construction

We begin by developing a mathematical description of the structuring primitives commonly found in DSMLs. Initially, this description, called a *domain*, shall be independent from metamodeling. Intuitively, the domain of a DSML classifies all the structurally legal (well-formed)

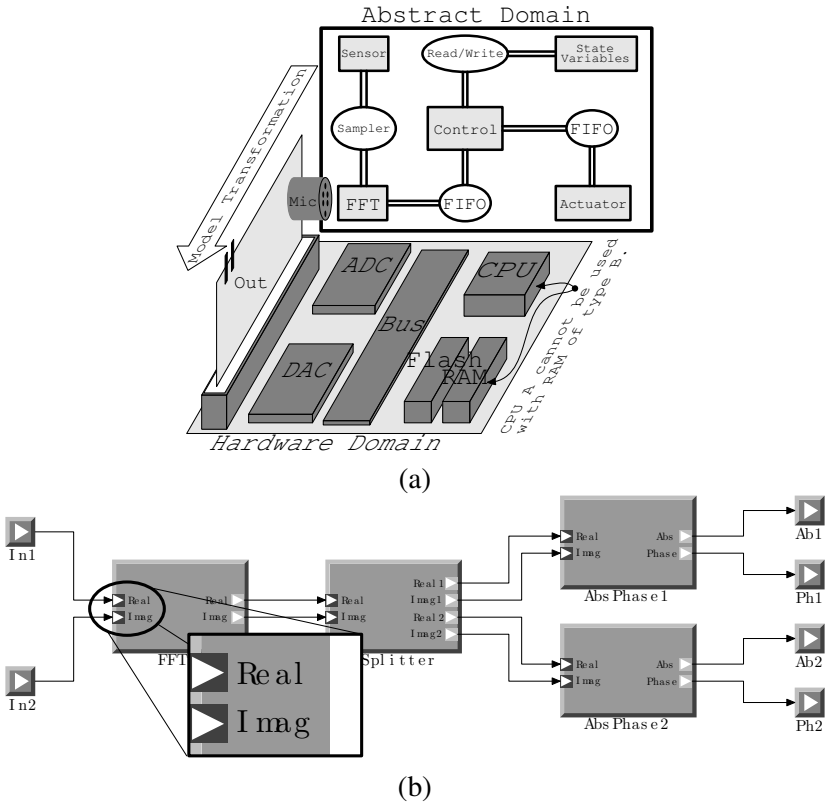


Fig. 1. (a) Example of two domains. (b) A model from a digital signal processing domain.

objects[1] of that language. As an example, consider the circuit board labeled *Hardware Domain* in Figure 1.a. This circuit board has a structural semantics where the language primitives are the the hardware components (ASICs and cards) that can be plugged into the circuit board. Each slot in the circuit board encodes a restriction on the actual ASIC that can be placed in a particular location. For example, the block labeled `CPU` encodes the constraint that a CPU, not a RAM module, must be placed at that point. Constraints can be more complicated than simple placement rules. For example, Figure 1.a also requires that if a CPU of type `A` is placed on the board, then a RAM module of type `B` cannot be placed on the board. A *model realization* is a description that has no remaining degrees of freedom, e.g., every place on the circuit board has some hardware assigned to it. In order to avoid confusion, we shall use the term model realization to indicate some construction that may or may not satisfy the placement rules. A *well-formed model realization* is a model realization that satisfies all the constraints imposed on its construction. If a model realization is well-formed then we refer to it simply as a *model*. Of course, this usage of the term model is with respect to the structural semantics. We do not need to give any details about what CPUs, RAMs, and buses do in order to check well-formedness of the circuit boards. A domain gives the necessary information to decide which model realizations are well-formed and which are not.

The primitives and placement rules can be applied to more abstract contexts. The plane labeled *Abstract Domain* in Figure 1.b uses functionally abstract primitives as the building blocks of the language. For example, infinite FIFOs, controllers, and sensors are the indivisible blocks. The FIFOs illustrate relational primitives that must exist between other parts of the model realization. Conceptually, this view of a model realization is the one taken by *platform-based design* [85]. It is a useful conceptualization because it includes relational constructs and constraints without implying a metamodel or OOP-like instance semantics. We use this intuition to formalize the information needed to characterize the domain of a DSML. A domain is specified by the following information:

1) Some mathematical structure Υ of concepts, components, or primitives from which model realizations are built,
2) a set R_Υ of all possible model realizations,
3) a set of domain constraints C over R_Υ.

[1]The term *object* is used with the categorical connotation in mind, i.e. a category has objects [63]. We are not implying an instance semantics.

The set R_Υ captures all the possible ways that language primitives can be composed. The set of *well-formed models realizations* is the set of all model realizations that satisfy the domain constraints. One might wonder why the set of domain constraints is treated as a first-class concept. Would it be simpler to imagine a set R_Υ^{well} consisting of just well-formed model realizations? In practice the only finite descriptions of the well-formed sets are constraint-based descriptions. This can be seen in classical context-free languages. Take $\Upsilon = \Sigma$ to be a finite alphabet, and $R_\Upsilon = \Sigma^*$ to be the set of all finite strings over Σ. The domain constraints C are specified by a BNF grammar or an automaton that accepts only well-formed strings.

The complex structuring found in DSMLs demands a flexible form for Υ and a rich mechanism for generating R_Υ. Ideally, Υ should easily capture the commonly used structural primitives, which can be fairly complex for the DSMLs. For example, *associations*, which are relational, can be *contained* within some substructure. UML containment is also a type of association, illustrating that relations over relations are common. We can see from graph theory that some new machinery is required even for the simplest case of *hierarchical graphs*, where nodes contain subgraphs [21].

A. Encoding Structuring Primitives

We provide this flexibility by selecting Υ to be a finite signature and R_Υ to be the powerset of the *term algebra* over Υ generated by an alphabet Σ. This description uses the language of *universal algebra*; see [11] for detailed discussion of term algebras. We now present an example to illustrate this approach. This example will build the necessary intuition for those not familiar with universal algebra. Figure 1.b shows a model realization representing a digital signal processing (DSP) system. We will work backwards from this system to the domain of all DSP models. To begin, we must extract the primitive concepts used to build DSP systems.

Examining Figure 1.b, we see that the system has inputs and outputs at the far left and right side, as well as a number of DSP primitives (FFT, phase/magnitude extraction, and signal demultiplexing), which can used multiple times. The zoomed-in box shows that primitives have interfaces, which are sets of uniquely identifiable *ports*. (Note this is not a toy example. Systems such as these have been extensively studied [56].) In order to capture these structuring concepts, we will describe a set of n-ary *function symbols* for encoding the modeling concepts. Informally, function symbols are place-holders for functions. We use them to encode information without attaching a deeper behavioral interpretation. (Again, see [11] for more details.) Table I lists the basic concepts of the DSP domain written as n-ary function symbols.

$$\Upsilon = \begin{cases} insig(X) : \texttt{X is system-wide input signal} \\ outsig(X) : \texttt{X is system-wide output signal} \\ prim(X) : \texttt{X is a basic DSP operation} \\ iport(X,Y) : \texttt{X has an input port called Y} \\ oport(X,Y) : \texttt{X has an output port called Y} \\ inst(X,Y) : \texttt{X is an instance of the DSP operation Y} \\ flow(X_1,Y_1,X_2,Y_2) : \texttt{Data goes from oport } Y_1 \texttt{ on } X_1 \texttt{ to iport } Y_2 \texttt{ on } X_2 \end{cases}$$

TABLE I

SET OF MODELING CONCEPTS FOR DSP DOMAIN.

These function symbols encode the essential structuring primitives of DSP systems. However, just as with "regular" functions, the symbols must be applied over some set of objects. These objects are understood as distinguishable items in the model realizations. For example, in Figure 1.b there is a DSP block called FFT. We capture this by writing $prim(\texttt{FFT})$, where the name FFT is a member some underlying alphabet. Mathematically, $prim(\texttt{FFT})$ is called a *term*; it is a combination of function symbols with alphabets. A model realization is a set of terms where each term expresses information about some particular members of the alphabet by applying the function symbols. Table II shows a partial encoding of the DSP model as terms[2]. Notice that terms can arbitrarily nested, naturally expressing relations over relations.

Primitives	$prim(\texttt{FFT})$, $prim(\texttt{Splitter})$, $prim(\texttt{Phase})$
Ports	$iport(prim(\texttt{FFT}),\texttt{Real}),\ldots,$ $oport(prim(\texttt{FFT}),\texttt{Imag})$
Inputs	$insig(\texttt{In1})$, $insig(\texttt{In2})$
Outputs	$outsig(\texttt{Ab1}), outsig(\texttt{Ph1}),\ldots,outsig(\texttt{Ph2})$
Instances	$inst(\texttt{FFT},prim(\texttt{FFT})),\ldots,$ $inst(\texttt{AbsPhase1},prim(\texttt{Phase}))$
Flows	$flow(insig(\texttt{In1}),insig(\texttt{In1}),inst(\texttt{FFT},$ $prim(\texttt{FFT})),iport(prim(\texttt{FFT}),\texttt{Real})),\ldots$

TABLE II

A PARTIAL ENCODING OF FIGURE 1.B WITH GROUND TERMS.

Working backwards, a model realization is a set of terms, therefore the set of all model realizations R_Υ contains all possible sets of terms that can be formed from Υ and an (infinite) alphabet Σ. We will make this more precise using the language of algebra. Assume an underlying vocabulary $\mathcal{V}$ providing function symbols, then Υ is a *signature* – a partial function from function

[2]In order to simplify the encoding, we assume that every input/output is also a port with the same name as the input/output.

names to the non-negative integers: $\Upsilon : \mathcal{V} \rightarrow \mathbb{Z}_+$. The domain of Υ is the subset of function symbols from $\mathcal{V}$ used to encode model realizations; the integer assigned to each symbol is the corresponding arity of the function. An Υ-algebra $\mathbf{A} = \langle A, \Upsilon \rangle$ is a structure where A is a set called the *universe* of the algebra, and Υ is a signature. Each function symbol f in the signature denotes a mapping $f : A^{\Upsilon(f)} \rightarrow A$ from an $\Upsilon(f)$-tuple of the universe back to the unverse.

Given a signature Υ and an alphabet Σ, there exists a special algebra $T_\Upsilon(\Sigma)$ called the *term algebra over* Υ *generated by* Σ. The precise definition of the term algebra is rather technical, so we defer this exposition to [11]. Instead, we emphasize the elegance of the term algebra as a construction for R_Υ. Definition III.1 shows how all the terms can be constructed using algebraic induction.

Definition III.1. Let Υ be a signature and Σ be an alphabet, then the *terms* of the term algebra can be inductively generated.

1) $\sigma \in \Sigma$ is a term
2) If $f \in \mathbf{dom}\ \Upsilon$ and $t_1, t_2, \ldots, t_{\Upsilon(f)}$ are terms, then $f(t_1, t_2, \ldots, t_{f(\Upsilon)})$ is a term.

Terms have *unique readability*[3], meaning that two terms are the same if and only if they are written exactly the same way. For example, $prim(\texttt{FFT}) \neq prim(\texttt{FFT1})$ because the arguments of $prim$ differ between the terms. Term equality can be easily checked, and we use this throughout the paper. Let $T_\Upsilon(\Sigma)$ denote the term algebra over Υ generated by Σ.

Definition III.2. The set of model realizations R_Υ for a signature Υ is:
$R_\Upsilon = \mathcal{P}\left(T_\Upsilon(\Sigma) \right)$.

B. Characterizing the Models

The set R_Υ contains many model realizations, some of which do not represent meaningful constructions in the DSML. For example, the output of signal processing block should not be wired to the output of another block, yet R_Υ includes realizations with this property. Erroneous realizations can be eliminated by imposing additional constraints over R_Υ, in the same way as OCL constraints are added to metamodels.

We formalize constraints using a proof-theoretic approach, which allows domains to be analyzed using proof procedures. This general framework requires the selection of a logic L for writing constraints. Let $\mathcal{F}(L)$ denote the class of all formulas of the logic L and $\Gamma_L(\Psi)$ denote the *deductive closure* of a set of formulas $\Psi \subset \mathcal{F}(L)$. The deductive closure contains all the formulas (finitely) deduced from Ψ using the logic L. It is also convenient to write $\Psi \vdash_L \phi$ to indicate that ϕ is deducible from Ψ, i.e. $\phi \in \Gamma_L(\Psi)$.

The domain constraints are a finite set of formulas Θ from the logic L. These formulas will be used to deduce if a model realization satisfies the well-formedness criteria of a particular domain. The procedure is as follows:

1) Convert a model realization r (finite set of terms) into a formula $\Psi(r)$,
2) Calculate the deductive closure $\Gamma_L(\Psi(r) \cup \Theta)$,
3) Examine the deductive closure to find if r satisfies the rules of the domain.

This framework requires minimal assumptions about the logic L. Namely, that terms can be converted to formulas and the logic contains some form of conjunction. Formally, there exists a one-to-one inclusion map $\iota : \mathcal{T}_\Upsilon \rightarrow \mathcal{F}(L)$ assigning a term to a formula of the logic. Conjunction, denoted $\psi \wedge \phi$, means

$$\forall \psi, \phi \in \mathcal{F}(L) \quad (\psi \wedge \phi) \vdash_L \psi \text{ and } (\psi \wedge \phi) \vdash_L \phi. \tag{III.1}$$

$$\forall \psi, \phi \in \mathcal{F}(L) \quad \{\psi, \phi\} \vdash_L (\psi \wedge \phi). \tag{III.2}$$

Given a logic with these simple properties, the procedure to decide well-formedness of a model realization is: Check if some agreed upon formulas are deducible. Or, check if some agreed upon formulas are not deducible. It is important to allow both of these styles, as not all logics are closed under negation. We call the domains using the former style *positive* and domains using the latter style *negative*.

Definition III.3. A realization $r = \{t_1, t_2, \ldots, t_n\}$ satisfies domain constraints Θ of a positive domain if:

$$\exists w \in T_W, \quad \iota(t_1), \iota(t_2), \ldots, \iota(t_n), \Theta \vdash_L w \tag{III.3}$$

where T_W is an agreed upon set of formulas used to witness the well-formedness of r.

Definition III.4. A realization $r = \{t_1, t_2, \ldots, t_n\}$ satisfies domain constraints Θ of a negative domain if:

$$\forall m \in T_M, \quad \iota(t_1), \iota(t_2), \ldots, \iota(t_n), \Theta \nvdash_L m \tag{III.4}$$

where T_M is an agreed upon set of formulas used to witness the malformedness of r.

For the remainder of the paper let $\Psi(r) = \{\iota(t_1), \iota(t_2), \ldots, \iota(t_n)\}$.

The connection between terms and formulas via ι admits a simple mechanism for constructing positive and negative domains. Add a new function symbol

[3]Notice that unique readability does not hold in most algebras: $x + y = +(x, y) = +(y, x) = y + x$.

$wellform(\cdot)$ to the signature Υ, then a model realization r is well-formed if $\exists x \in \mathcal{T}_\Upsilon\ \Psi(r), \Theta \vdash \iota(wellform(x))$. In the case of negative domains augment Υ with a $malform(\cdot)$ symbol. A model realization is well-formed if $\forall x \in \mathcal{T}_\Upsilon\ \Psi(r), \Theta \nvdash \iota(malform(x))$, i.e. if its impossible to deduce any $malform(\cdot)$ term from r. Given that terms can be converted to formulas, we may write that "*a term t is derived from a realization r*" as a shorthand for $\exists \phi \in \mathcal{F}(L),\ \iota(t) = \phi$ and $\Psi(r), \Theta \vdash_L \phi$.

A *domain* has the following parts: An alphabet Σ, a signature Υ, called the *domain signature*, a signature Υ_C, called the *constraint signature*, and a set of constraints C for deriving well-formedness. Υ_C, an extension of Υ, contains all the necessary symbols for deriving well-formedness. By "extension", we mean that Υ_C contains at least the symbols of Υ, while assigning the same arity to common symbols: $\forall f \in \mathbf{dom},\ \Upsilon(f) = \Upsilon_C(f)$. Domains are subdivided into two disjoint classes: *positive* and *negative*. Positive domains must include the unary symbol $wellform(\cdot)$ in Υ_C; negative domains must include the unary function symbol $malform(\cdot)$ in Υ_C.

Definition III.5. A domain D is a 4-tuple of the form $\langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$. Υ and Υ_C are signatures with functions symbols from $\mathcal{V}$, where Υ_C is an extension of Υ. Σ is an alphabet, and C is a finite set of formulas of $\mathcal{F}(L)$. A domain is *positive* if $\Upsilon_C(wellform) \mapsto 1$; it is *negative* if $\Upsilon_C(malform) \mapsto 1$.

In our framework a domain is the *structural semantics* of the a DSML. This scheme permits a simple definition of equivalence between domains. In the interest of space, we show this for positive domains.

Lemma III.6. *Let $models(D)$ be the set of well-formed model realizations of the domain D. There exists an equivalence relation $\cong$ over positive domains:*

$$D_1 \cong D_2 \text{ if } models(D_1) = models(D_2) \qquad \text{(III.5)}$$

It is easy to see that this is a valid equivalence relation, so we omit the proof. Enforcing a common symbol $wellform(\cdot)$ makes it possible to compare arbitrary domains. Realistically, domain equivalence can be computed by examining the ways in which $wellform(\cdot)$ terms are deduced. Figure 2 illustrates how two domains can be compared with each other. Consider the case of two positive domains D_1, D_2 with identical Υ signatures and agreeing[4] constraint signatures Υ_C. The upper plane shows the well-formed models of each domain as a subset of the powerset of terms. Each well-formed model in D_i infers one or more terms $wellform(x)$ under the consequence operator $\vdash_L$, as shown on the lower plane. Let T_W be all the formulas corresponding to terms of the form $wellform(x)$. Two domains can be compared by working backwards from T_W in the following way: Check if there exists a model realization $r \in R_\Upsilon$ such that $\exists w \in T_W,\ \Psi(r), C_1 \vdash_L w$, but $\neg \exists u \in T_W,\ \Psi(r), C_2 \vdash_L u$. If there exists such an r, then the domains cannot be equal because r is well-formed in the first domain and not in the second. If there does not exist such an r, then $models(D_1) \subseteq models(D_2)$. In this case, check the opposite direction for an r' such that $\neg \exists w \in T_W,\ \Psi(r'), C_1 \vdash_L w$ and $\exists u \in T_W,\ \Psi(r'), C_2 \vdash_L u$. Again, if no such r' can be found, then $models(D_1) = models(D_2)$. A similar comparison can be made for negative domains. If an appropriate style of logic is selected (i.e. an appropriate consequence relation) then domain equivalence is decidable.

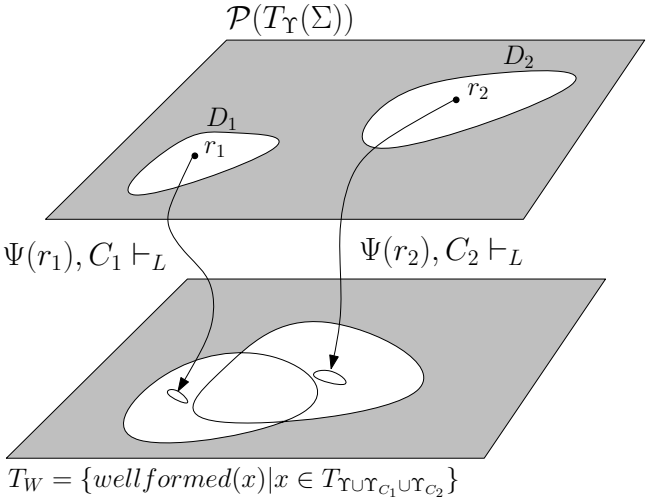


Fig. 2. Two positive domains of same signature are related by the $wellform$ symbol

Theorem III.7. *Two positive domains D_1, D_2 are equivalent iff there is no model realization $r \in R_{\Upsilon_i}$ such that*

$$\exists w \in T_W,\ \Psi(r), C_i \vdash_L w \text{ and } \forall u \in T_W,\ \Psi(r), C_j \nvdash_L u$$

for $i = 1, j = 2$ and $i = 2, j = 1$.

This theorem is a refutation proof against the inclusions of Lemma III.6. If neither implication can be refuted, then the domains must be equivalent. Though this paper is not about theorem proving techniques, we have developed a solver, FORMULA [51], that implements analysis on domains.

Figure 3 shows a simplified version of a modeling problem that we gave to a class of computer science students. The metamodel of 3.a contains an abstract class

[4]The signatures assign the same arity to symbols of the same name.

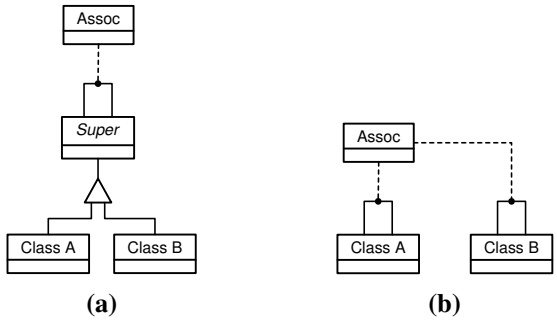


Fig. 3. (a) Instances of *Class A* can be associated with instances of *Class B* due to inheritance. (b) Instances of *Class A* cannot be connected to instance of *Class B*.

Super and two sub-classes: *Class A* and *Class B*. An association class called *Assoc* can connect instances of type *Super*. The students had to determine if an "equivalent" metamodel could be formed by removing the abstract super class, and propagating the association down the inheritance hierarchy. Figure 3.b shows the metamodel that results from this operation. In this simplified example it is easy to see that the two metamodels are not equivalent, because in the modified metamodel *Class A* instances cannot be connected to *Class B* instances. We can use Theorem III.7 to prove that this is the case. The first step is to extract corresponding domains from each metamodel. (Section III-D covers this extraction process in detail.) After the domain definitions have been extracted, Theorem III.7 is applied.

The first step in the proof tries to construct a model of domain D_b (from metamodel b) that is not well-formed in D_a. Not surprisingly, there are few feasible solutions to the goal and the proof fails. Thus, either $models(D_b) \subset models(D_a)$ or $models(D_b) = models(D_a)$. The second proof tries to refute the inclusion $models(D_a) \subseteq models(D_b)$. This proof goal yields a satisfiable solution tree and the prover constructs a solution refuting this inclusion, so it must be that $models(D_b) \subset models(D_a)$. In fact, the prover constructs a concrete model containing an instance of *Class A* and an instance of *Class B* connected by an association of type *Assoc*. This concludes the proof that $D_a \ncong D_b$.

C. Model Transformations

Model transformations translate the syntactic constructs of one domain into the syntactic constructs of another domain. Transformations between DSMLs provide the foundation for a number of key modeling activities. These activities include:

1) raising and/or lowering the degree of abstraction. For example, *Model Driven Architecture* (MDA) uses model transformations to provide a bridge between high-level implementation-independent models and less abstract implementation-dependent models. (These are called platform-independent and platform-dependent models in MDA.)
2) the specification of DSML *behavioral* semantics. A purely structural domain without behavioral semantics can inherit a behavioral semantics via a transformation to a well-defined DSML.
3) high-level and analyzable code generators [73].

In the most general sense, a model transformation tool implements a mapping from model realizations to model realizations. We call such a mapping an *interpretation*.

Definition III.8. An *interpretation* $[\![\ ]\!]$ is a mapping from the model realizations of a domain D to the model realizations of another domain D'.

$$[\![\ ]\!] : R_\Upsilon \mapsto R_{\Upsilon'} \tag{III.6}$$

A single domain may have many different interpretations, and these form a family of mappings $([\![\ ]\!]_j)_{j\in J}$. For some model realization $r \in R_\Upsilon$, we denote the j^{th} interpretation of r as $[\![r]\!]_j$. Mathematically, interpretations are general enough that they can be used to specify arbitrary relationships between domains. For example, an interpretation from domain D_X onto *StateCharts* [40] attaches a hierarchical concurrent automata semantics to D_X. Similar strategies have been used to affix continuous time [62], dataflow [67], and hybrid dynamics [42] to domains. Thus, we will view a *domain specific modeling language* (DSML) as a domain and its family of interpretations.

Definition III.9. A *domain specific modeling language* (DSML) L is a pair comprised of its domain and interpretations.

$$L = \langle\ D,\ ([\![\ ]\!]_j)_{j\in J}\ \rangle\,. \tag{III.7}$$

Admittedly, not all interpretations are easily specified, however the model transformation community has shown that many important interpretations can be specified by sets of transformation rules. A transformation rule takes the form $L \rightarrow R$ where L and R are the left- and right-hand side of the rule, respectively. The left-hand side of the rule is a pattern that is matched against the input model realization. The right-hand side is a pattern that is instantiated and combined with the output model realization whenever L matches. A model transformation generates an output model by applying the rules until no more rules can be applied. See [18] for a survey of

existing model transformation techniques. We wish to incorporate model transformations into the algebraic/logic framework of domains. This is accomplished by specifying transformations rules as formulas that deduce the elements of the output model realization from the input model realization. This approach is similar in spirit to the declarative subset of the VIATRA tool suite [17].

Definition III.10. A *transformation* T is a three tuple:

$$T = \langle \Upsilon, \Upsilon', \tau \rangle \tag{III.8}$$

where Υ, Υ' are disjoint signatures, and τ is a set of formulas of the same logic L used to specify constraints.

A model realization $r \in R_\Upsilon$ is transformed to a model realization $r' \in R_{\Upsilon'}$ by converting deduced formulas to terms. A transformation T defines a *transformational interpretation*.

Definition III.11. Given a transformation T, a *transformational interpretation* $[\![\ ]\!]^T$ is a mapping:

$$[\![\ ]\!]^T : R_\Upsilon \to R_{\Upsilon'}, \tag{III.9}$$

$$[\![r]\!]^T \mapsto \{s \in \mathcal{T}_{\Upsilon'} \mid \iota(s) \in \Gamma_L(\Psi(r) \cup \tau)\}. \tag{III.10}$$

A transformational interpretation first finds all the formulas that can be deduced from the input r and the transformation clauses τ. The output realization r' is built from the deduced formulas that also correspond to terms in the $\mathcal{T}_{\Upsilon'}$.

We now present an example that characterizes the asynchronous (shuffle) product of two finite state automata (FSAs) using transformational interpretations. We use a style of logic, called *Horn logic*, that is described in the next section. The input domain contains function symbols for encoding two parallel automata, and the output domain contains function symbols for encoding a single product automaton. For example:

$$\Upsilon = \left\{ \begin{array}{l} s_1(x), s_2(x), initial_1(x), initial_2(x), \\ event_1(x), event_2(x), e_1(x, \alpha, y), e_2(x, \alpha, y) \end{array} \right. \tag{III.11}$$

Symbols of the form f_1 (or f_2) are used by the first (or second) automaton. For example, the $s_1(x)$ symbol encodes the states of the first automaton, while the $s_2(x)$ symbol encodes the states of the second automaton. The $initial_i$ symbols encode the initial states of the automata, and the $event_i$ symbols encode the event alphabets of the automata. A transition is of the form $e_i(x, \alpha, y)$ indicating that automaton i transitions from state x to state y on event α. Similarly, the function symbols of the product automaton are:

$$\Upsilon' = \left\{ \begin{array}{l} s_{12}(x, y), initial_{12}(x, y), \\ event_{12}(x), e_{12}(x, \alpha, y) \end{array} \right. \tag{III.12}$$

The transformation formulas τ explain how to deduce the product automaton from the two concurrent automata. For example, we know that the event alphabet of the product automata is the union of the event alphabets of the smaller automata:

$$event_{12}(x) \leftarrow event_1(x)$$
$$event_{12}(x) \leftarrow event_2(x)$$

Informally, these formulas state that whenever there is a term $event_i(\mathtt{c})$ in the input, a term $event_{12}(\mathtt{c})$ is added to the output, for $\mathtt{c} \in \Sigma$. Effectively, they union the two event alphabets. Similarly, the (initial) states of the product automata are the Cartesian product of the (initial) states of the smaller automata:

$$s_{12}(x, y) \leftarrow s_1(x), s_2(y)$$
$$initial_{12}(x, y) \leftarrow initial_1(x), initial_2(y)$$

The asynchronous product represents the situation that two concurrent automata never transition at the same time. For example, if there is a transition $s \xrightarrow{\alpha} s'$ in the first automaton, then the product automaton contains a transition $(s, t) \xrightarrow{\alpha} (s', t)$ for every state t in the second automaton. Again, this rule is described with formulas:

$$e_{12}(s_{12}(x, t), \alpha, s_{12}(y, t)) \leftarrow e_1(x, \alpha, y), s_2(t)$$
$$e_{12}(s_{12}(t, x), \alpha, s_{12}(t, y)) \leftarrow s_1(t), e_2(x, \alpha, y)$$

Finally, the conversion from terms to Horn formulas is simple:

$$\forall t \in \mathcal{T}_\Upsilon \ \iota(t) \mapsto (t \leftarrow true) \tag{III.13}$$

A term becomes a *fact*, which is a Horn clause asserting t to be deducable.

Figure 4.a shows two concurrent automata. These automata would form the following set of terms (model realization):

$$r = \left\{ \begin{array}{l} s_1(\mathtt{A}), s_1(\mathtt{B}), s_1(\mathtt{C}), s_2(\mathtt{D}), s_2(\mathtt{E}), \\ initial_1(\mathtt{A}), initial_2(\mathtt{D}), \\ event_1(\mathtt{e}_1), event_1(\mathtt{e}_2), event_2(\mathtt{e}_3), \\ e_1(\mathtt{A}, \mathtt{e}_1, \mathtt{B}), e_1(\mathtt{A}, \mathtt{e}_2, \mathtt{C}), e_2(\mathtt{D}, \mathtt{e}_3, \mathtt{E}) \end{array} \right. \tag{III.14}$$

Take τ to be the previous formulas, then

$$[\![r]\!]^T = \left\{ \begin{array}{l} s_{12}(\mathtt{A}, \mathtt{D}), s_{12}(\mathtt{A}, \mathtt{E}), s_{12}(\mathtt{B}, \mathtt{D}), \\ s_{12}(\mathtt{B}, \mathtt{E}), s_{12}(\mathtt{C}, \mathtt{D}), s_{12}(\mathtt{C}, \mathtt{E}), \\ initial_{12}(\mathtt{A}, \mathtt{D}), event_{12}(\mathtt{e}_1), \\ event_{12}(\mathtt{e}_2), event_{12}(\mathtt{e}_3), \\ e_{12}(s_{12}(\mathtt{A}, \mathtt{D}), \mathtt{e}_1, s_{12}(\mathtt{B}, \mathtt{D})), \\ e_{12}(s_{12}(\mathtt{A}, \mathtt{D}), \mathtt{e}_2, s_{12}(\mathtt{C}, \mathtt{D})), \\ e_{12}(s_{12}(\mathtt{A}, \mathtt{D}), \mathtt{e}_3, s_{12}(\mathtt{A}, \mathtt{E})), \\ e_{12}(s_{12}(\mathtt{B}, \mathtt{D}), \mathtt{e}_3, s_{12}(\mathtt{B}, \mathtt{E})), \\ e_{12}(s_{12}(\mathtt{A}, \mathtt{E}), \mathtt{e}_1, s_{12}(\mathtt{B}, \mathtt{E})), \\ e_{12}(s_{12}(\mathtt{A}, \mathtt{E}), \mathtt{e}_2, s_{12}(\mathtt{C}, \mathtt{E})), \\ e_{12}(s_{12}(\mathtt{C}, \mathtt{D}), \mathtt{e}_3, s_{12}(\mathtt{C}, \mathtt{E})) \end{array} \right. \tag{III.15}$$

Figure 4.b shows the resulting product automaton described by these terms.

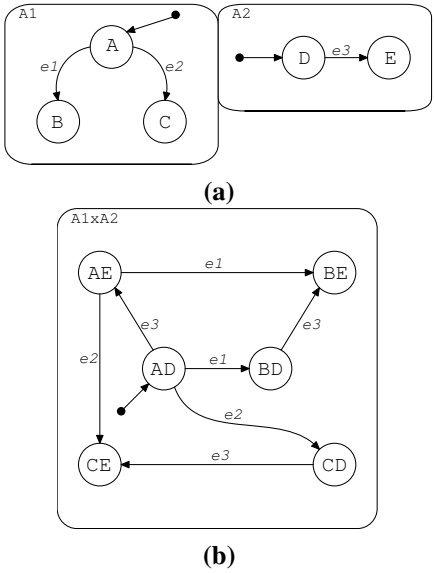


Fig. 4. (a) Two concurrent automata (b) Asynchronous product of two automata.

Model transformations are defined within the same mathematical framework as domains. This enables analysis of the interaction between transformations and domain constraints. For example, interpretations that preserve the well-formedness rules of domains are particularly important to embedded system design. These *structure preserving maps* posses the weakest property that one would expect a correct transformational semantics to posses, and are important in correct-by-construction design [43] [20] [7].

Definition III.12. Let $D = \langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$ and $D' = \langle \Upsilon', \Upsilon'_C, \Sigma, C' \rangle$ be domains. Let $T = \langle \Upsilon, \Upsilon', \tau \rangle$ be a transformation. The transformational interpretation $[\![\ ]\!]^T$ is a *structure preserving map* from D to D' if:

$$\forall r \in R_\Upsilon \Big(r \in models(D) \Rightarrow [\![r]\!]^T \in models(D') \Big). \tag{III.16}$$

Even the verification of weak properties is still a major open problem in the model transformation community. Our approach allows some of these properties to transcribed into formal logic and then proved with an existence proof. This is accomplished by first renaming the constraint function symbols so they are disjoint: $(\Upsilon_C - \Upsilon) \cap (\Upsilon'_C - \Upsilon') = \emptyset$. In particular, we create two distinct well-formedness symbols $wellform(\cdot) \in \Upsilon_C$ and $wellform'(\cdot) \in \Upsilon'_C$. Similarly, the constraint formulas C and C' are rewritten to use the renamed function symbols. Call the renamed signatures $\overline{\Upsilon_C}, \overline{\Upsilon'_C}$ and the renamed formulas $\overline{C}, \overline{C'}$. (This renaming scheme can be done automatically. See [51] for a description of how FORMULA implements this renaming scheme.)

Given two domains D, D' and a compatible transformation T, then the renamed formulas force the domains to interact only through the model transformation. Mathematically we look at the combined formulas: $\overline{C} \cup \overline{C'} \cup \tau$. If there exists an input model realization r such that $\exists x,\ \Psi(r), \overline{C}, \overline{C'}, \tau \vdash_L \iota(wellform(x))$, but $\forall y,\ \Psi(r), \overline{C}, \overline{C'}, \tau \nvdash_L \iota(wellform'(y))$, then T is not structure preserving. However, if no such r exists then T is structure preserving. Again, this techniques tries to refute the implication of Equation III.16

Theorem III.13. *Let D, D', and T be two positive domains and a transformation as described in Definition III.12. Then T is structure preserving iff:*

$$\begin{array}{l} \nexists r \in R_\Upsilon\ \exists x \in \mathcal{T}_{\overline{\Upsilon_C}}(\Sigma)\ \forall y \in \mathcal{T}_{\overline{\Upsilon'_C}}(\Sigma) \\ \Psi(r), \overline{C}, \overline{C'}, \tau \vdash_L \iota(wellform(x)), \\ \Psi(r), \overline{C}, \overline{C'}, \tau \nvdash_L \iota(wellform'(y)) \end{array} \tag{III.17}$$

A similar theorem exists for negative domains.

Theorem III.14. *Let D, D', and T be two negative domains and a transformation as described in Definition III.12. Then T is structure preserving iff:*

$$\begin{array}{l} \nexists r \in R_\Upsilon\ \exists y \in \mathcal{T}_{\overline{\Upsilon_C}}(\Sigma)\ \forall x \in \mathcal{T}_{\overline{\Upsilon'_C}}(\Sigma) \\ \Psi(r), \overline{C}, \overline{C'}, \tau \nvdash_L \iota(malform(x)), \\ \Psi(r), \overline{C}, \overline{C'}, \tau \vdash_L \iota(malform'(y)) \end{array} \tag{III.18}$$

We now illustrate this proof technique with a concrete example. A deterministic FSA is an automaton such that there is no state with two different transitions guarded by the same event. We can define the domain of deterministic concurrent automata by adding the following malformedness rules:

$$C = \begin{cases} malform(x) \leftarrow \\ e_1(x, \alpha, y), e_1(x, \alpha, y'), y \neq y'. \\ malform(c) \leftarrow \\ e_2(x, \alpha, y), e_2(x, \alpha, y'), y \neq y'. \end{cases} \tag{III.19}$$

The determinism rule can also be defined for product automata.

$$C' = \begin{cases} malform(s_{12}(x, y)) \leftarrow \\ e_{12}(s_{12}(x, y), \alpha, s_{12}(x', y')), \\ e_{12}(s_{12}(x, y), \alpha, s_{12}(x'', y'')), \\ x' \neq x''. \\ malform(s_{12}(x, y)) \leftarrow \\ e_{12}(s_{12}(x, y), \alpha, s_{12}(x', y')), \\ e_{12}(s_{12}(x, y), \alpha, s_{12}(x'', y'')), \\ y' \neq y''. \end{cases} \tag{III.20}$$

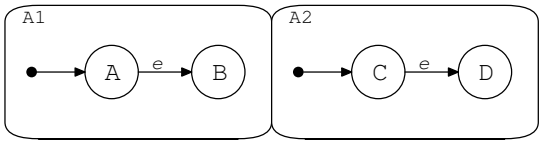


Fig. 5. Generated counter-example showing that shuffle product does not preserve determinism.

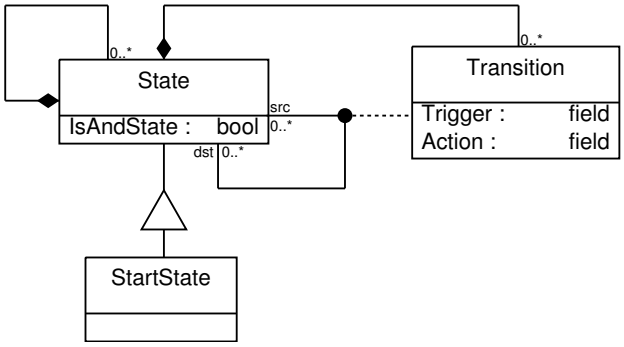


Fig. 6. MetaGME metamodel for HFSM.

We can now prove that the shuffle product is not structure preserving with respect to these domains. In another words, we prove that the shuffle product does not preserve determinism. Of course, this fact is well-known; the novelty is our reformulation of the problem as a question of structure preservation between domains. The proof engine generates a counter example showing two deterministic automata that yield a non-deterministic product. Figure 5 shows this counter example. This example shows that by unifying domains and model transformations we can apply theorem proving techniques to reason about their composition. Note that this solution is generated independently from the examples of Figure 4; only the domain constraints are used.

D. Metamodels and Metamodeling

Domains and interpretations provide the basic foundations for model-based design through DSMLs. In this section we formalize more advanced DSML design principles using our formalization as a foundation. Specifically, we formalize the *metamodeling process* by which new domains are rapidly defined via the construction and interpretation of *metamodels*. A metamodel is a model that belongs to a special DSML called a *metamodeling language*. The metamodeling language provides an interpretation that maps metamodels to domains. This process allows users to concisely "model" their domain, and then generate the domain concepts and constraints from the model.

Domains are characterized by UML-like class diagrams called *metamodels*. Figure 6 shows a metamodel of a hierarchical automata domain. The boxes in the metamodel are the primitive concepts of the language. Superficially, the boxes appear to be classes. However, this is merely a historical artifact inherited from UML notation, which has given these boxes the nickname of "class". It is important to distinguish metamodel classes from true classes, which are much more than structuring primitives. True classes have behavioral semantics describing, among other things, how the flow of control changes when a method is invoked or when static constructors are called. Nonetheless, the "class" metaphor is convenient and allows metamodels to provide a compact notation for specifying structure. For example, language primitives can contain data members (called *attributes*): The `Transition` class has `Trigger` and `Action` attributes, both of type `field` (or string). The metamodel also encodes a graph class by associating some classes with vertices and other classes with edges. The `State` and `StartState` classes correspond to vertices; instances of the `Transition` class are edges. The diagram also declares which vertex types can be connected together, and gives the edge types that can make these connections. The solid lines passing through the connector symbol (•) indicate that edges can be created between vertices, and the dashed line from the connector to the `Transition` class indicates that these edges are instances of type `Transition`. The diagram encodes yet more rules: Lines that end with a diamond (⧫) indicate hierarchical containment, e.g. `State` instances can contain other states and transitions. Lines that pass through a triangle (△) identify inheritance relationships, e.g. a `StartState` inherits the properties of `State`.

This example illustrates two important points about metamodeling languages. First, a small metamodel can define a rich domain that may include a non-trivial inheritance hierarchy, a graph class, and other concepts like hierarchical containment and aspects. Metamodels are concise specifications of complex domains. Second, the *meanings* of metamodeling constructs are tedious to define, and the language appears idiosyncratic to users. This problem is compounded by the fact that competing metamodeling languages are "defined" with excessively long standards: The GME manual [44], much of which is devoted to metamodeling, is 224 pages. The Meta Object Facility (MOF) language, an OMG standard used by MDA and UML, requires a 358 page description [75]. These long natural language descriptions mean that tool implementations are likely to differ from the standards, and that the standards themselves are more likely to be inconsistent or ambiguous.

We hope to alleviate some of these problems by

formalizing the metamodeling process. We present a novel approach to metamodeling semantics by using domains and transformational interpretations as the building blocks to define metamodeling. A metamodeling language L_{meta} is a DSML with a special interpretation $[\![\ ]\!]_{meta}$ (called the *metamodeling semantics*) that maps *models* to *domains*:

$$L_{meta} = \langle D_{meta}, ([\![\ ]\!]_{meta}) \rangle \tag{III.21}$$

The domain D_{meta} captures the structure of metamodels. The transformational interpretation $[\![r]\!]_{meta}$ maps metamodel realizations r to a new domains. There is one technical caveat: Interpretations, as we have defined them, map model realizations of one domain to models realizations of another domain. In order to make a mapping from models to domains, we need to create a *domain of domains* that provides a structural encoding for domains. A domain of domains is created by constructing a special domain $D_{\mathcal{F}}$ capable of representing formulas from $\mathcal{F}(L)$. Formally, $D_{\mathcal{F}}$ is paired with a bijection $\delta : \mathbb{Z}_+^{\mathcal{V}} \times \mathbb{Z}_+^{\mathcal{V}} \times \mathcal{P}(\mathcal{F}(L)) \to D_{\mathcal{F}}$ that maps two signatures and a set of formulas to a model in the special domain $D_{\mathcal{F}}$. The notation $\mathbb{Z}_+^{\mathcal{V}}$ is the set of all partial functions from $\mathcal{V}$ to $\mathbb{Z}_+$, i.e. the set of all signatures. Note that for the domain of domains we will fix a particular Σ.

This approach allows us to specify metamodeling languages transformationally, as shown in Figure 7. The domain D_{meta} represents a metamodeling language with some arbitrary notation for describing domains (e.g. UML). T_{meta} is a transformation that converts models in D_{meta} to a structural representation of a domain in $D_{\mathcal{F}}$. The transformation T_{meta} encodes the semantics of the metamodeling language. For example, the metamodel r_m is transformed to the structural representation d_m of a domain by applying the transformational interpretation $[\![r_m]\!]^{T_{meta}}$. The actual domain defined by a metamodel is recovered by the inverse function δ^{-1} that recovers a domain from a structural representation in $D_{\mathcal{F}}$. Thus, the domain defined by the metamodel r_m is discovered by applying $\delta^{-1}([\![r_m]\!]^{T_{meta}})$. Our formalization also allows us to describe the notion of *metacircularity* precisely. Intuitively, a metamodeling language is metacircular if there exists a metamodel in the language that defines the language. Formally, a metamodeling language is metacircular if there exists a well-formed metamodel r_{mm} such that $D_{meta} \cong \delta^{-1}([\![r_{mm}]\!]^{T_{meta}})$. The metamodel r_{mm} is called the *meta-metamodel*, as shown in Figure 7. This can be imagined geometrically: The set of all well-formed metamodels forms a decision boundary in $R_{\Upsilon_{meta}}$. A metamodeling language is metacircular if there exists a metamodel that reconstructs the decision boundary of the metamodeling language.

Finally, our view of metamodeling bares resemblance to the construction of the *category of small categories* in category theory [63]. The main differences are:

1) Domains are intentionally concrete, and so they are not as general as arbitrary categories. Equivalently, the class of domains can be viewed a sub-class of small categories.
2) We will formalize $[\![\ ]\!]^{T_{meta}}$ and $D_{\mathcal{F}}$ so that they are described within a mathematical framework supporting constructive theorem proving.
3) We provide an explicit mechanism for describing domains via the objects of a single domain (D_{meta}).

We point this out for the readers interested in category theory. There are certainly interesting categorical views that arise from this work, though these are outside the scope of this paper.

E. Applications of Structural Semantics

We have provided a structural semantics for DSMLs, which formalizes a fundamental set of modeling activities. For the remainder of this paper we examine how our structural semantics can be applied to existing modeling tools. This is important for several reasons: First, by applying our semantics to a mature modeling framework, we show that our mathematical apparatus is sufficiently powerful to describe current modeling practice. Second, we can use a concrete formalization to link modeling tools with the analysis techniques presented earlier in this section. For example, after formalizing a particular metamodeling language of some tool, we can extract tool-independent characterizations of domains. These domains can be analyzed, composed, transformed, used for design-space exploration, etc... In order to make this more than just an exercise, we apply our formalism to the well-known *Model-Integrated Computing* (MIC) tool suite [58].

IV. Formalizing Model-Integrated Computing

The MIC tool suite is a tried-and-tested DSML-based modeling framework that has evolved over a decade of development. Its primary components are a metamodeling language called *MetaGME*, a metaprogrammable modeling environment called the *Generice Modeling Environment* (GME), and a feature-rich model transformation framework called the *Graph Rewriting And Transformation* (GReAT) engine. The MIC tool suite has been used to model complex and diverse systems

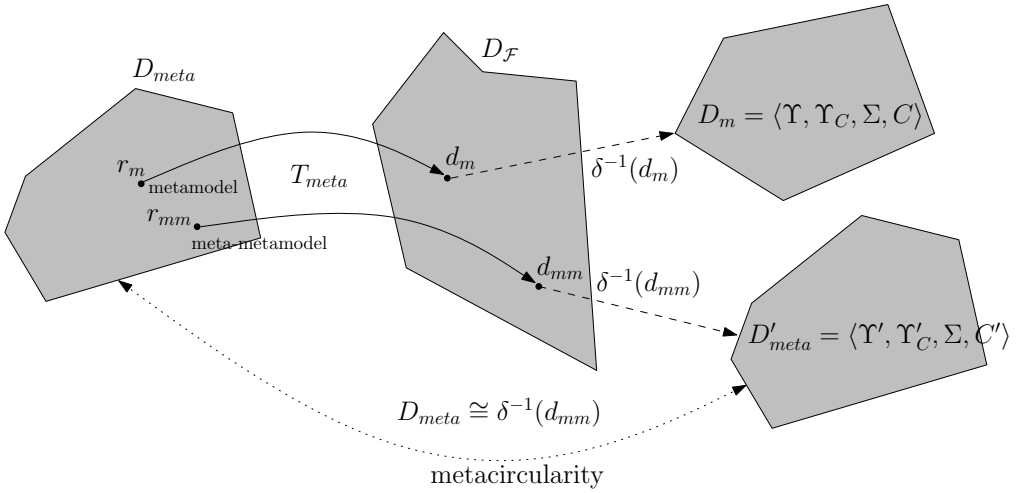


Fig. 7. Abstract view of the metamodeling process

ranging from the software systems of the NASA space station to the sensor and data fusion networks of the Saturn automobile plant. In order to formalize the MIC tool suite, we must create a concrete instantiation of our structural semantics for each component within MIC. Additionally, we must create an interface from the MIC tools to the formal definitions. For example, it should be possible to read a metamodel created with MetaGME and extract the corresponding domain definition. Similarly, it should be possible to read a model constructed within GME and check its conformance against a tool-independent domain definition. In this section, we illustrate the formalization process for a subset of the MetaGME metamodeling language. In the interest of space, we do not present the formalization of GReAT, though we do mention how such a formalization fits into the overall picture.

We now present a formalized and tool-independent MIC framework called *MiniMeta*. As the name implies, MiniMeta is a scaled-down version of MIC. MiniMeta uses a simplified metamodeling language called *MiniMOF*, which is similar to the OMG's own scaled-down metamodeling language called *essential meta-object facility* (eMOF) [77]. In this sense, MiniMOF can be viewed as capturing the core features found among metamodeling languages. MiniMeta also provides a model transformation language called *MiniGReAT*, though we do not describe MiniGReAT here. However, during the formalization of MiniMeta it becomes apparent that MiniGReAT reuses all of the framework, except for one component that would have to be defined from scratch. This reuse is possible because we have defined domains and transformations within the same mathematical framework. Generating a transformation from a *transformation model* is not a significantly different process than generating a domain from a metamodel.

A key parameter of our structural semantics is the style of logic used to express constraints and transformations. We select a form of *Horn logic with negation* to specify constraint and transformation formulas in MiniMeta. This logic is both expressive and supports constructive theorem proving [49]. The next subsection describes this style of logic and discusses its expressiveness. After selecting the logic, we must construct a "domain of Horn domains" $D_{\mathcal{H}}$ that contains a model for each domain definition with constraints as Horn formulas. This special domain is coupled with the structural representation function δ, which is a bijective map from domain definitions to models of $D_{\mathcal{H}}$.

Figure 8 shows the architecture of the MiniMeta framework. The MiniMOF language contains two parts: the domain D_{meta} and a transformational interpretation $[\![\ ]\!]_{meta}$. D_{meta} characterizes the syntax of metamodels. For example, the upper-left blue circle is a metamodel r_m, which is a set of terms from the term algebra of D_{meta}. Assuming r_m satisfies the constraints of D_{meta}, then $[\![r_m]\!]_{meta}$ yields a Horn model h_m. This Horn model is the domain defined by r_m, but encoded as a model realization of $D_{\mathcal{H}}$. In order to recover the actual domain, δ^{-1} is applied to h_m resulting in the domain $D_m = \langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$. A similar procedure occurs for MiniGReAT, except that MiniGReAT has its own syntax for transformations (D_{trans}) and its own transformational interpretation onto $D_{\mathcal{H}}$ ($[\![\ ]\!]_{trans}$). Recall that δ^{-1} yields two signatures and a set of Horn formulas. In the case of transformations, we interpret the two signatures as the source and target signatures Υ, Υ', instead of the model and constraint signatures Υ, Υ_C. Similarly, the formulas are interpreted as the set of transformation formulas τ, instead of the constraint axioms C. We have

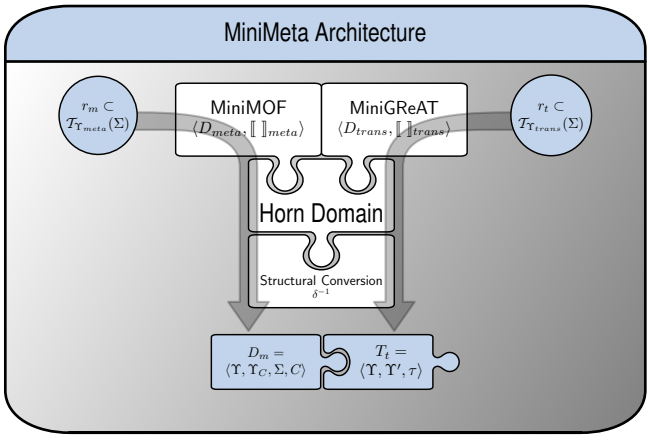


Fig. 8. The architecture for the MiniMeta Framework.

implemented this in a recent toolset called *BAM* [50].

The MiniMeta architecture builds a metamodeling and a model transformation facility using the fundamental concepts of domains and model transformations. Furthermore, all of the components of MiniMeta are specified within a compatible mathematical framework. There is only one point in the process where we leave the mathematical framework of term algebras and consequence operators; this occurs at the conversion from Horn models to domains, i.e. within δ. However, we shall show by example that δ is sufficient simple as to not complicate analysis of the overall framework. Finally, the framework results in domain and transformation definitions that are independent from the languages used to specify them. The blue puzzle pieces in Figure 8 represent a domain and transformation generated via this process. The generation process yields stand-alone entities, which can be further composed.

A. Review of Horn Logic, Extensions, and Expressiveness

First, let us review some basic definitions, beginning with basic Horn logic. *Formulas* are built from terms with *variables* and logical *connectives*. There are different approaches for distinguishing variables from constants. One way is to introduce a new alphabet Σ_v that contains variable names such that $\Sigma \cap \Sigma_v = \emptyset$. The terms $\mathcal{T}_{\Upsilon_C}(\Sigma)$ are called *ground terms*, and contain no variables. This set is also called the *Herbrand Universe* denoted $\mathcal{U}_H$. The set of all terms, with or without variables, is $\mathcal{T}_{\Upsilon_C}(\Sigma \cup \Sigma_v)$, denoted $\mathcal{U}_T$. Finally, the set of all *non-ground* terms is just $\mathcal{U}_T - \mathcal{U}_H$. A *substitution* ϕ is term endomorphism $\phi : \mathcal{U}_T \to \mathcal{U}_T$ that fixes constants. In another words, if a substitution ϕ is applied to a term, then the substitution can be moved to the inside $\phi f(t_1, t_2, \ldots, t_n) = f(\phi t_1, \phi t_2, \ldots, \phi t_n)$. A substitution does not change constants, only variables, so $\forall g \in \mathcal{U}_H,\ \phi(g) = g$. We say two terms $s, t \in \mathcal{U}_T$ *unify* if there exists a substitution ϕ that make the terms identical $\phi s = t$, and of finite length. (This implies the *occurs check* [88] is performed.) We call ϕ the unifier of s and t. The variables that appear in a term t are $vars(t)$, and the constants are $const(t)$.

A *Horn clause* θ is a formula of the form $h \leftarrow t_1, t_2, \ldots, t_n$ where h is called the *head* and $t_1, \ldots, t_n$ are called the *tail* (or body). We write T_θ to denote the set of all terms in the tail. The head only contains variables that appear in the tail, $vars(h) \subseteq \bigcup_i vars(t_i)$. A clause with an empty tail ($h \leftarrow$) is called a *fact*, and contains no variables. Recall that these clauses will be used *only* to calculate model properties. This is enforced by requiring the heads to use those function symbols that do not encode model structure, i.e. every head $h = f(t_1, \ldots, t_n)$ has $f \in (\Upsilon_C - \Upsilon)$. (Proper subterms of h may use any symbol.) This is similar to restrictions placed on declarative databases [71].

We slightly extend clauses to permit *disequality* constraints. A Horn clause with disequality constraints has the form $h \leftarrow t_1, \ldots, t_n, (s_1 \neq s'_1), (s_2 \neq s'_2), \ldots, (s_m \neq s'_m)$, where s_i, s'_i are terms with no new variables $vars(s_i), vars(s'_i) \subseteq \bigcup_i vars(t_i)$. We can now define the *meaning* of a Horn clause. The definition we present incorporates the *Closed World Assumption* which assumes all conclusions are derived from a finite initial set of facts I. Given a set of Horn clauses Θ, the operator $\widehat{\Gamma}$ is called the *immediate consequence operator*, and is defined as follows:

$$\widehat{\Gamma}(X, \Theta) = X \cup \left\{ \phi(h_\theta) \;\middle|\; \begin{array}{c} \exists \phi, \theta,\ \phi(T_\theta) \subseteq X \text{ and} \\ \forall (s_i \neq s'_i)_\theta \in \theta,\ \phi s_i \neq \phi s'_i \end{array} \right\} \tag{IV.1}$$

where X is a set of facts known facts, ϕ is a substitution, and θ is a clause in Θ. Notice that the disequality constraints force the substitutions to keep certain terms distinct. The deductive closure of Horn logic is calculated by repeatedly applying Γ until no new facts are derived. *Nonrecursive Horn logic* adds the restriction that the clauses of Θ can be ordered $\theta_1, \theta_2, \ldots, \theta_k$ such that the head h_{θ_i} of clause θ_i does not unify with any tail $t \in T_{\theta_j}$ for all $j \leq i$. This is a key restriction; without it, the logic can become undecidable. Consider the recursive clause $\Theta = \{f(f(x)) \leftarrow f(x)\}$. Then $\{f(\mathtt{c}_1)\} \vdash_\Theta \{f(\mathtt{c}_1), f(f(\mathtt{c}_1)), \ldots, f(f(f(\ldots f(\mathtt{c}_1) \ldots)))\}$ includes an infinite number of distinct terms. The transformation clauses for the shuffle product and the constraints for deterministic automata are instances of this non-recursive Horn logic.

Pure Horn logic imposes too much of a restriction on the structure of domains. We can extend Horn logic by introducing a pseudo-negation that is commonly called *negation-as-failure* (NAF). This extension allows terms in the tail to be "negated", e.g. $h \leftarrow \neg t$, which is interpreted as h can be proved if t cannot be proved. Thus, negation is equivalent to the failure of inference of certain terms. It turns out that this small change has a resounding affect on the corresponding theory, placing it in realm of non-monotonic logic [13] [30]. In the interest of space, we shall leave the reader with this informal description of negation-as-failure.

Finally, we allow *term restrictions* to be placed on domains. A term restriction forces all well-formed models to use a finite set of terms of the form $f(\ldots)$ that are explicitly enumerated. For example, if a domain has a signature $\Upsilon = \{(f,1),(g,2)\}$, and we wish to place term restrictions on f, then we may write $\forall m \in models(D),\ \{f(x)|f(x) \in m\} \subseteq \{f(\mathtt{c}_1), f(\mathtt{c}_2)\}$. This restriction indicates that if a model m is well-formed then every term of the form $f(x) \in m$ has either $x = \mathtt{c}_1$ or $x = \mathtt{c}_2$ for $\mathtt{c}_1, \mathtt{c}_2 \in \Sigma$. We will simplify this notation by writing $\mathbf{restrict}(f, \{f(\mathtt{c}_1), f(\mathtt{c}_2)\})$. Note that Horn logic has already been implemented in programming languages like Prolog [16], usually with the SLD resolution algorithm [2]. For simple problems, like checking model well-formedness, we can directly use these existing tools. However, most of the analysis problems we encounter (e.g. domain equivalence) require more sophisticated tools. The theorem prover FORMULA was developed for analyzing DSMLs. Prolog also includes an implementation of NAF, but it must be used carefully as it may be unsound. However, we haven taken care to ensure that our descriptions can be soundly evaluated by Prolog implementations of NAF.

The expressiveness of nonrecursive Horn logic with negation determines the domains and transformations that can be expressed within this incarnation of our structural semantics. In turns out that it is non-trivial to quantify the expressiveness of this extended form. Since this logic is non-monotonic, it cannot be described as a simple subset of first order logic (which is monotonic) as we normally expect. A more effective way to measure the expressiveness is via complexity theory, i.e. by the class of problems that have polynomial time reductions to a set of extended Horn formulas. Non-monotonicity can be formalized in different ways, and this yields various complexity results. However, for the nonrecursive case the following result typically holds: Given a term t, an initial set of terms I, and set of clauses Θ determining if $I, \Theta \vdash t$ is PSPACE-complete under *stratified negation*. This complexity class includes the NP-complete problems and the problem of membership in context-sensitive languages. See [19] for an in depth discussion of complexity results in logic programming. These results show that nonrecursive Horn logic extended with negation provides an expressive foundation.

B. Defining the MiniMOF Domain

In this section we formalize the domain of MiniMOF metamodels, D_{meta}. The MiniMOF notation is based on the Unified Modeling Language (UML), which is also standardized by the OMG. This notation supports the following essential concepts: classes, associations, attributes, containment, and inheritance. Since UML is usually drawn in a graph-like notation, we will imagine that metamodel syntax is composed from vertex-like and edge-like concepts. (The actual encoding may be more complicated than just unary and binary relations.) Table III lists the vertex-like primitives. The first column describes the primitives and any rules dictating the use of those primitives. The second column depicts a typical use-case of the primitives using UML-like notation. Table IV provides similar data for the edge-like primitives.

These tables describe how primitives are composed into metamodels, but they do not describe how metamodels encode domains. This will be defined transformationally in Section IV-D. At this point we are only describing the constraints on metamodel structure; this is done by characterizing the malformed metamodels using the $malform$ symbol. Some constraints are easy to describe: For example, *attribute*s have a *type* field indicating the data type of the attribute. This field must take one of the following values: $\{\mathtt{bool}, \mathtt{string}, \mathtt{enum}\}$. Such a constraint is specified by placing a term restriction on a unary $type$ symbol, where each term of the form $type(\cdot)$ indicates a legal type.

$$\mathbf{restrict}\left(type, \left\{ \begin{array}{c} type(\mathtt{bool}), \\ type(\mathtt{string}), type(\mathtt{enum}) \end{array} \right\}\right)$$

The term restriction states that every occurrence of $type(x)$ must have $x \in \{\mathtt{bool}, \mathtt{string}, \mathtt{enum}\}$. The *incidence* property on an *association endpoint* is another example of a term restriction – An association endpoint can be either a source or destination.

A more interesting constraint comes from the acyclic nature of inheritance, or, more precisely, from the nature of cycles themselves. The constraint that an inheritance hierarchy has *no cycles*, is actually an infinite list of Horn constraints: *no self-loops*, *no cycles of length two*, *no cycles of length three*,..., ad infinitum. For analysis purposes the logic should be acyclic (no recursion), but this prevents a faithful rendering of properties with an infinite number of distinctly different (non-isomorphic) forms. The reader familiar with logic programming might recall that cycle checking can be easily specified in

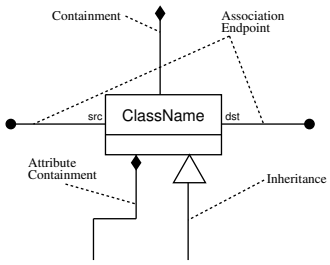
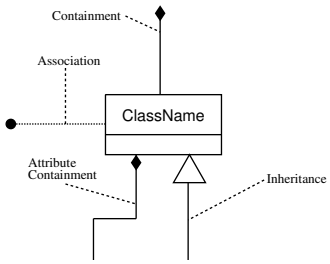
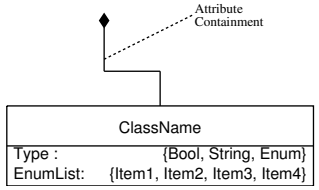
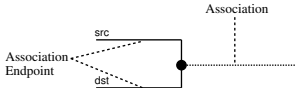
Vertex Primitives	MiniMOF
Class A class is primitive part of a metamodel that can have *Containment*, *Attribute Containment*, *Association Endpoint*, *Inheritance* edges incident on it. Every class has a *name*.	
Association Class An association class is a primitive on which a *Containment*, *Association*, *Attribute Containment*, and *Inheritance* edges can be incident. Every association class has a *name*.	
Attribute Class An attribute class is a primitive on which *Attribute Containment* edges can be incident. Every attribute class has a *name* and a *type* which can be *boolean*, *string*, or *enumeration*. Attribute classes of type enumeration may have a list of *enumeration values*.	
Connector A connector has exactly two *Association Endpoint* edges and one *Association* edge incident on it.	

TABLE III
TABLE OF VERTEX PRIMITIVES FOR MINIMOF METAMODELS

Prolog via a combination of lists and recursion. However, this increase in expressiveness leads to undecidability [9]. Thus, the only way to properly encode an acyclic inheritance hierarchy is to approximate the *"no cycles"* constraint for a finite range of cycle lengths from 1 to n. Let the symbol $inheritance(x, y)$ denote an inheritance edge from x to y. This auxiliary symbol is placed in the constraint signature Υ_C. Similarly, let $ipath_3(x, y, z) \in \Upsilon_C$ and $ipath_4(x, y, z, w) \in \Upsilon_C$ indicate directed inheritance paths of length three and four. The following axioms calculate inheritance paths of these lengths:

$$ipath_3(x, y, z) \leftarrow inheritance(x, y), inheritance(y, z), (x \neq y \neq z)$$

$$ipath_4(x, y, z, w) \leftarrow ipath_3(x, y, z), inheritance(z, w), (w \neq x \neq y \neq z)$$

An inheritance path of length three is made up of two inheritance edges across three distinct vertices. The disequality constraints $x \neq y \neq z$ require the vertices in the path to be distinct. Three such constraints are needed to ensure that all three vertices are distinct. The four-path is defined by the presence of a three-path and a new edge that extends the three-path by one unique vertex. This process can be continued to define any path of finite length. A cycle of length $n > 2$ is defined by the presence of a path of length n and an inheritance edge that connects the end of the path to the beginning. The definitions for $icycle_1$, $icycle_2$, $icycle_3$ and $icycle_4$ are shown below:

$$\begin{aligned}
icycle_1(x) &\leftarrow inheritance(x, x)\\
icycle_2(x, y) &\leftarrow inheritance(x, y), (x \neq y)\\
icycle_3(x, y, z) &\leftarrow ipath_3(x, y, z), inheritance(z, x)\\
icycle_4(x, y, z, w) &\leftarrow ipath_4(x, y, z, w), inheritance(w, x)
\end{aligned}$$

Finally, an inheritance hierarchy is malformed

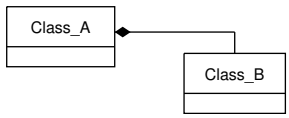
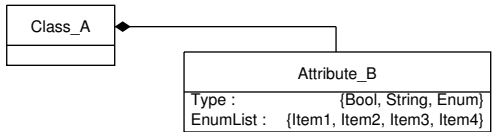
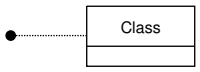
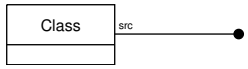
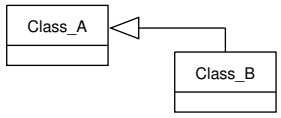
Edge Primitives	MiniMOF
Containment An edge that must terminate on a *Class*.	
Attribute Containment An edge that must start on an *Attribute Class*.	
Association An edge that must start on a *Connector* and end on an *Association Class*.	
Association Endpoint An edge that must start on a *Connector* and end on an *Association Class*. Association endpoints have a *incidence* that can be either *source* or *destination*.	
Inheritance An edge that cannot form a directed cycle consisting only of inheritance edges.	

TABLE IV
TABLE OF EDGE PRIMITIVES FOR MINIMOF METAMODELS

($imalform$) if there is any such cycle.

$$
\begin{aligned}
imalform(icycle_1(x)) &\leftarrow icycle_1(x)\\
imalform(icycle_2(x,y)) &\leftarrow icycle_2(x,y)\\
imalform(icycle_3(x,y,z)) &\leftarrow icycle_3(x,y,z)\\
imalform(icycle_4(x,y,z,w)) &\leftarrow icycle_4(x,y,z,w)
\end{aligned}
$$

The remaining constraint axioms for MiniMOF are listed in Appendix III. Table VI describes the function symbols used to encode the vertex-like primitives and lists the constraint axioms associated with each primitive. Table VII provides the same information for edge-like primitives. These tables constitute a complete and tool-independent specification of the legal MiniMOF metamodels. However, before metamodels can be transformed into domains, a precise definition of the *domain of Horn domains* ($D_{\mathcal{H}}$) must be constructed.

C. $D_{\mathcal{H}}$ - The Domain of Horn Domains

In this section we formalize the domain of Horn domains. In order to accomplish this, we fix some parameters:

1) There is a fixed vocabulary $\mathcal{V}$ of function symbols. The signatures of every domain have function symbols from $\mathcal{V}$.
2) There is a fixed alphabet Σ. Every domain uses this alphabet to form the term algebra $\mathcal{T}_\Upsilon(\Sigma)$.
3) The fixed alphabet contains the positive integers $\mathbb{Z}_+ \subset \Sigma$.

Furthermore, there exists some subset of Σ, called Σ_F, that is bijectively related to the vocabulary of function symbols of $\mathcal{V}$, via a bijection δ_f. This bijection allows a function symbol to be translated into a constant, for the purpose of representation. A similar bijection δ_V must exist between a subset of Σ, called Σ_V, and the set of variable names used by the class of Horn formulas $\mathcal{F}_{Horn}$. Choose Σ so that $\Sigma_F \cap \Sigma_V = \emptyset$.

Given these parameters, we must construct a domain $D_{\mathcal{H}} = \langle \Upsilon^{\mathcal{H}}, \Upsilon^{\mathcal{H}}_C, \Sigma, C^{\mathcal{H}} \rangle$ with models that represent the two signatures and constraints of domains specified with Horn axioms. This must be done so that there exists a bijection δ, called the *structural representation function*, that maps a domain X to a model $\delta(X) \in models(D_{\mathcal{H}})$. The inverse map δ^{-1} recovers the domain

from a model of $D_{\mathcal{H}}$. Table V (Appendix III) lists the function symbols of $\Upsilon^{\mathcal{H}}$ that encode these elements, as well as the constraints $C^{\mathcal{H}}$ on the composition of these symbols. The table also informally describes the result of inverting each term in a Horn model back into a element of a domain definition, via δ^{-1}.

There are a few constraints not listed in Table V that cannot be defined within the restricted Horn logic of Section IV-A. These three constraints require additional operators that calculate the successor of an integer ($+1$) and compute subterms ($\sqsubseteq$). The first constraint requires the arity of a $map_n(x,\ldots)$ term to match the arity of the function symbol x over which map is applied.

$$malform(map_n(x,y_1,\ldots,y_{n-1})) \leftarrow map_n(x,y_1,\ldots,y_{n-1}), def(t,x,y), (n \neq y+1)$$

The second constraint requires that variables introduced in the head of a clause must have been introduced in the tail of the clause. To express this constraint we introduce a *subterm* relation $\sqsubseteq$ such that a term t' is a subterm of a term t if t' appears in an argument of t or an argument of some subterm of t.

$$vargood(v,x) \leftarrow axiom(x,h), tail(x,t), (var(v) \sqsubseteq h), (var(v) \sqsubseteq t)$$
$$malform(axiom(x,h)) \leftarrow axiom(x,h), (var(v) \sqsubseteq h), \neg vargood(v,x)$$

The final constraint prohibits negation in the head of a clause, as negation does not have meaning in the head.

$$malform(axiom(x,h)) \leftarrow axiom(x,h), (neg(h') \sqsubseteq h)$$

Though these axioms are not written in the strict Horn logic that we previously defined (Section IV-A), they do not significantly impact algorithms that deduce formal properties of domains. In fact, this gap can be closed by extending $\Upsilon^{\mathcal{H}}$ to encode these two additional operators. However, we omit this in the interest of space.

Let $\mathcal{F}_{Horn}$ be the set of all Horn formulas. The relationship between Horn models and domains is formalized by a mapping $\delta : \mathbb{Z}_+^{\mathcal{V}} \times \mathbb{Z}_+^{\mathcal{V}} \times \mathcal{P}(\mathcal{F}_{Horn}) \rightarrow models(D^{\mathcal{H}})$ from a domain (signatures and constraints) to a model of the Horn domain. This mapping is defined by a structural induction over
an arbitrary domain $D = \langle \Upsilon, \Upsilon_C, \Sigma, C \rangle$ (See Definition IV.1).

Definition IV.1. The structural representation function δ is given by the following induction:

1) $$\delta(\Upsilon, \Upsilon_C, C) \mapsto \delta(\Upsilon) \cup \delta(\Upsilon_C) \cup (\bigcup_{s \in C} \delta(s))$$

2) $$\delta(\Upsilon) \mapsto \bigcup_{f \in \mathbf{dom}\Upsilon} def(\texttt{sig}, \delta_f(f), \Upsilon(f))$$

3) $$\delta(\Upsilon_C) \mapsto \bigcup_{f \in \mathbf{dom}\Upsilon} def(\texttt{con}, \delta_f(f), \Upsilon(f))$$

4) $s_i \in C, s_i = \begin{pmatrix} H \leftarrow \neg L'_1, \ldots, \neg L'_m, \\ L_1, \ldots, L_n, (v_{j_1} \neq v_{j_2}), \\ \ldots, (v_{j_k} \neq v_{j_k}) \end{pmatrix}$

$$\delta(s_i) \mapsto \begin{cases} axiom(i, \delta(H)) \cup \\ \bigcup_{L' \in s_i} tail(i, neg(\delta(L')) \cup \\ \bigcup_{L \in s_i} tail(i, \delta(L)) \cup \\ \bigcup_{(v \neq u) \in s_i} tail(i, neq(\delta(v), \delta(u))) \end{cases}$$

5) $\delta f(t_1, t_2, \ldots, t_n) \mapsto map_{\Upsilon(f)+1}(\delta_f(f), \delta(t_1), \ldots, \delta(t_n))$
6) $\delta(x) \mapsto var(\delta_v(x))$ where x is a variable. $\delta(\texttt{c}) \mapsto \texttt{c}$ where $\texttt{c}$ is a constant.

Domains viewed as two signatures and a family of axioms, have an equivalence ($=$) between them that only takes into account the equivalence of their notation: $D_i = D_j$ if $\Upsilon_i = \Upsilon_j$, $\Upsilon_{C_i} = \Upsilon_{C_j}$, and $C_i = C_j$. This equivalence is a weaker form of equivalence that requires both sets of constraints to be written in exactly the same way. It holds that $(D_i = D_j) \Rightarrow (D_i \cong D_j)$, but the converse does not hold.

Lemma IV.2. *Fix Σ, $\mathcal{V}$, δ_f, and δ_V. The representation function δ is a one-to-one and onto function from domains with Horn axioms to the set of well-formed Horn models $D_{Horn}(\Upsilon_{Horn}, C_{Horn})$.*

We briefly sketch the proof. By our construction, the variables and function symbols of a domain are bijectively mapped to members of Σ. From this encoding, it is clear that $\delta^{-1}(\delta(X)) = X$, which establishes that δ is one-to-one:

$$\begin{aligned} \delta(X) = \delta(Y) \Rightarrow \\ \delta^{-1}(\delta(X)) = \delta^{-1}(\delta(Y)) \Rightarrow \\ X = Y \end{aligned}$$

Also, it is clear that a horn model $m_X \in models(D^{\mathcal{H}})$ encodes some domain X such that $\exists X,\ \delta(X) = m_X$. Thus, δ is one-to-one and onto, so it is a bijection and the inverse δ^{-1} is also a bijection.

D. MiniMOF Transformation onto D_{Horn}

We now described the transformation T_{meta} from MiniMOF metamodels to Horn models: $T_{meta} = \langle \Upsilon^{MiniMOF}, \Upsilon^{\mathcal{H}}, \tau_{meta} \rangle$. For the purposes of illustration, we will make this transformation as simple as possible: Every *class* named x in the input metamodel becomes a unary function symbol $x(n)$ in the Horn model, where the single argument n indicates that an object called n is an instance of x. For example, the automata metamodel in Figure 6 contains a class called `state` (i.e. the term $class(\texttt{state})$). This term will

be translated to a term in the Horn model that adds a unary function symbol $state$ to the domain signature: $def(\texttt{sig}, \texttt{state}, 1)$. Similar to classes, ***association classes*** become ternary function symbols $x(n, s, d)$, where the first argument n is the name of association instance, the second argument s identifies the source of the association, and the third argument d identifies the destination of the association. Attribute classes become binary function symbols $x(c, v)$, where the first argument c identifies the object containing the attribute instance and the second argument v identifies the value of the attribute instance. The transformation contains the following clauses:

$$\tau_{meta} \supset \left\{ \begin{array}{c} def(\texttt{sig}, x, 1) \leftarrow class(x) \\ def(\texttt{sig}, x, 2) \leftarrow attribute(x, y) \\ def(\texttt{sig}, x, 3) \leftarrow assocClass(x) \end{array} \right.$$

Converting metamodeling elements to function symbols is the simple part of the transformation. The core of the transformation must produce domain constraints that faithfully implement the metamodel. For example, a model is malformed if it assigns an improper value to an enumeration attribute. In order to introduce this constraint, we generate a function symbol $enumvalue$, that contains all the enumeration values for all enumeration attributes using term restrictions. Additionally, for each enumeration attribute, we generate a constraint consisting of a head and two tail terms that requires each attribute instance to use one of the enumerated values. These are generated with the following transformation:

$$\begin{array}{l} def(\texttt{sig}, \texttt{enumvalue}, 2) \leftarrow attribute(x, \texttt{enum}). \\ axiom\left(\begin{array}{c} enum(x, y), map_3(\texttt{enumvalue}, x, y) \end{array} \right) \leftarrow \\ \qquad enum(x, y), attribute(x, \texttt{enum}). \\ axiom\left(\begin{array}{c} attribute(x, \texttt{enum}), \\ malform(map_3(x, var(\texttt{y}), var(\texttt{z}))) \end{array} \right) \leftarrow \\ \qquad attribute(x, \texttt{enum}). \\ tail\left(\begin{array}{c} attribute(x, \texttt{enum}), map_3(x, var(\texttt{y}), var(\texttt{z})) \end{array} \right) \leftarrow \\ \qquad attribute(x, \texttt{enum}). \\ tail\left(\begin{array}{c} attribute(x, \texttt{enum}), \\ neg(map_3(\texttt{enumvalue}, x, var(\texttt{z}))) \end{array} \right) \leftarrow \\ \qquad attribute(x, \texttt{enum}). \end{array}$$

Assume that the `IsAndState` attribute of Figure 6 is a enumeration containing the constants `andState` and `orState`. The metamodeling transformation would produce the following encoding of the this attribute in the Horn model:

$$\begin{array}{l} def(\texttt{sig}, \texttt{IsAndState}, 2),\ def(\texttt{sig}, \texttt{enumvalue}, 2), \\ axiom(enum(\texttt{IsAndState}, \texttt{andState}), \\ \quad map_3(\texttt{enumvalue}, \texttt{IsAndState}, \texttt{andState})), \\ axiom(enum(\texttt{IsAndState}, \texttt{orState}), \\ \quad map_3(\texttt{enumvalue}, \texttt{IsAndState}, \texttt{orstate})), \\ axiom(attribute(\texttt{IsAndState}, \texttt{enum}), \\ \quad malform(map_3(\texttt{IsAndState}, var(\texttt{y}), var(\texttt{z})))), \\ tail(attribute(\texttt{IsAndState}, \texttt{enum}), \\ \quad map_3(\texttt{IsAndState}, var(\texttt{y}), var(\texttt{z}))), \\ tail(attribute(\texttt{IsAndState}, \texttt{enum}), \\ \quad neg(map_3(\texttt{enumvalue}, \texttt{IsAndState}, var(\texttt{z})))) \end{array}$$

Finally, applying the inverse representation function yields the following axioms:

$$\begin{array}{c} \{(IsAndState, 2), (enumvalue, 2)\} \subset \Upsilon \\ \textbf{restrict}\left(\begin{array}{c} enumvalue, \\ \left\{ \begin{array}{c} enumvalue(\texttt{IsAndState}, \\ \texttt{andState}), \\ enumvalue(\texttt{IsAndState}, \\ \texttt{orState}) \end{array} \right\} \end{array} \right) \\ malform(IsAndState(y, z)) \leftarrow IsAndState(y, z), \\ \neg enumvalue(\texttt{IsAndState}, z) \end{array}$$

Appendix I lists some additional rules of the MiniMOF transformation. This completes the full formalization of the MiniMOF metamodeling facility. We would have to repeat a similar formalization for the graph transformation language MiniGReAT. First, the domain D_{GReAT} would be defined, and then a transformation T_{GReAT} would transform models to transformations via the Horn Domain. However, formalizing MiniGReAT does not require any new techniques, so we omit its description.

E. Implementing MiniMeta with GME/GReAT

MiniMeta is a formally specified and tool-independent framework that reflects much of the key functionality found in the MIC tool suite. In this section we implement an interface from MIC to MiniMeta. This has several advantages: First, the mature MIC tool suite can be used to describe MiniMeta objects. This is more convenient than writing signatures, axioms, and terms. Second, this illustrates that existing metaprogrammable tools can be used to host new frameworks, like MiniMeta, without rewriting tools. Third, system modelers can perform their usual modeling activities without expertise in formal methods. Nonetheless, formal specifications can be extracted from their work.

The first feature to implement is a model editor for constructing MiniMOF metamodels and for checking well-formedness of metamodels. GME already provides a metamodeling facility, called ***MetaGME***, that converts a metamodel into a domain-specific model editor.

We desire a domain-specific model editor for building MiniMOF metamodels, and this can be achieved by building a MetaGME metamodel of the MiniMOF domain. This is something like a *meta-metamodel*, but the meta-metamodel is written in a different metamodeling language than the one it is describing. Figure 10 shows the MetaGME metamodel of MiniMOF. The notation of MetaGME is similar to that of MiniMOF, so the informal meaning of this metamodel should match the reader's intuition. We will not attempt to prove that the MetaGME metamodel is equivalent to our description of MiniMOF, as MetaGME does not have such a formalization. Rather, this problem shall be circumvented completely.

Upon construction of the MiniMOF metamodel, a new model editor can be generated. Figure 11.a shows a MiniMOF metamodel built with the generated model editor. This particular metamodel describes a dataflow domain similar to that of Figure 1.b. Thus, GME can be used to construct MiniMOF-like objects. However, these GME objects must be linked to true MiniMOF metamodels (which are subsets of the MiniMOF term algebra). We accomplish this by extending GME with an extraction component that converts a MiniMOF metamodel within GME into the set of terms encoding that metamodel. The terms are loaded into an embedded Prolog engine along with the formal definition of the MiniMOF domain (transcribed into Prolog syntax). The SLD resolution procedure of Prolog is used to formally prove that the metamodel within GME is well-formed. Prolog works well for this task, because it only has to reason about a single model (i.e. a fixed set of terms), as opposed to an entire domain. Additionally, GME includes a COM-based extension mechanism, so it is simple to add this component to the model editor. Prolog also provides a foreign language interface, allowing the Prolog engine to be embedded into the extraction component. Figure 9 shows the implementation of the metamodel authoring facility. This architecture bridges the gap between the implementation of MIC and the formalization of MiniMeta, without reauthoring either framework.

Figure 11.b shows the analysis component applied to the MiniMOF metamodel in Figure 11.a. The *Translation to definite clauses* list shows the conversion of the MiniMOF metamodel into terms (also called *definite clauses* in Prolog). Each of the translated terms is added to the knowledge base of the Prolog engine. The proving engine is then queried to prove $malform(x)$. If this cannot be proved, then the metamodel realization is well-formed. If $malform(x)$ can be proved then the metamodel realization is malformed. In this case, all solutions are displayed to this user; each solution identifies some problem in the metamodel. Figure 11.b shows that the engine is unable to prove $malform(x)$, so the metamodel realization is well-formed. If the metamodel is modified by creating an inheritance cycle from `Interface` to `Input`, then the Prolog engine detects this error. Figure 12 shows how this inheritance cycle was correctly detected by the extraction component. This implementation illustrates that existing tools can be readily used to build a formal metamodel authoring facility. Additionally, our tool architecture introduces a new level of flexibility not found in the existing metaprogrammable tools, because the MiniMOF domain is not hard coded into GME and can be easily modified. Supporting competing metamodeling frameworks within the same tool set is an important issue raised in [25], and we address it in our implementation of MiniMeta with MIC.

The next part of the MiniMeta/MIC framework generates domain definitions and modeling environments from MiniMOF metamodels constructed within GME. This generation process creates three artifacts. First, a mathematical definition of the domain is generated. Second, a GME modeling environment is generated from the metamodel. Third, an extractor component is attached to the newly generated modeling environment. This component extracts a set of terms from model realizations constructed within the modeling environment, and checks these terms against the mathematical definition of the domain. Figure 13 shows this process in detail; the blue objects are generated by the framework. A MiniMOF metamodel m_X is constructed within the MiniMOF environment of GME. After m_X has been checked for correctness, the domain generation facility can be applied. The first step is a conversion of m_X into a set of terms that are loaded into the Prolog engine. See arrow `1` of Figure 13. The metamodeling transformation T_{Meta} (Section IV-D) is also loaded into the

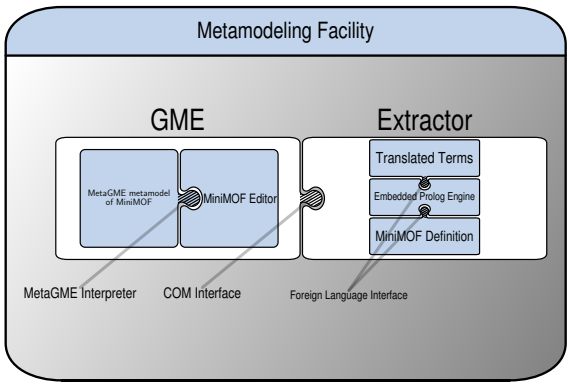


Fig. 9. Detailed view of metamodel authoring facility with MetaGME, GME, and and embedded Prolog Engine.

Prolog engine, and a Horn model h_X is calculated. The Horn model h_X is passed to the inverse representation function δ^{-1} that produces the domain definition in a pure mathematical notation and in Prolog notation. This process produces a tool-independent definition of domain X as defined by metamodel m_X.

In addition to the domain definition, the MIC tools can be used to generate a modeling editor for domain X. This accomplished by a GReAT transformation that converts the MiniMOF metamodel m_X into a MetaGME metamodel m'_X. GME reads this metamodel and generates a new modeling environment for domain X. Arrow 2 in the figure illustrates this process. Since we do not expect GME to enforce our formal MiniMeta semantics, there is no need to formalize the transformation from MiniMOF to MetaGME. At this point, a tool-independent definition of domain X has been generate and a tool-dependent modeling environment for domain X has been generated. The final step in the generation process creates a new extractor component that links these two entities. This is arrow 3 in the figure. The extractor component attaches to GME and converts domain X model realizations into set of terms that are checked against the formal definition generated in step 1. In this way, model realizations of domain X can be constructed and checked against the formal definition of X.

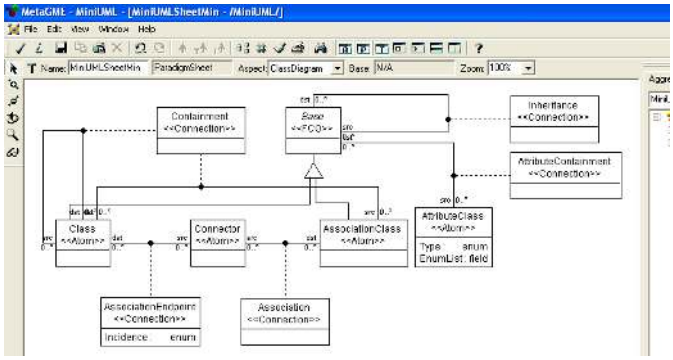
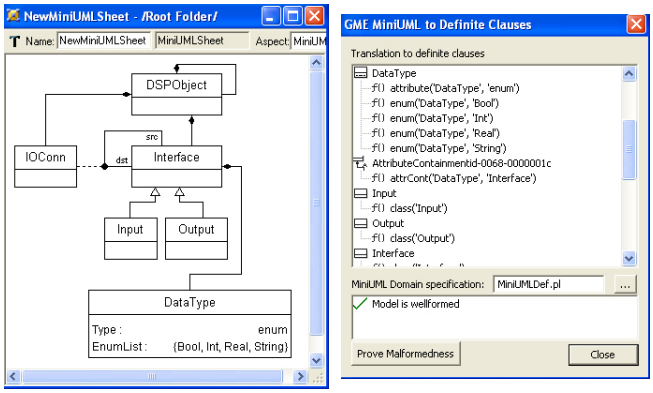


Fig. 10. A MetaGME metamodel of the MiniMOF Domain.

Figure 14 shows the domain generation component applied to the MiniMOF metamodel of Figure 11. The list labeled *Input Model* shows the translation of the DSP metamodel into terms. Below this list are options for selecting the types of objects generated by the component. If all of the options are checked, then the domain generator performs the entire process described above. The lists on the right-hand side show the terms in the generated Horn model. Appendix II shows the generated Prolog definition of the DSP domain after applying δ^{-1}.

After the domain generator completes, domain models



(a) (b)

Fig. 11. (a) A MiniMOF metamodel in GME of DSP domain. (b) Translation of metamodel to definite clauses and verification that metamodel is well-formed, using a Prolog engine.

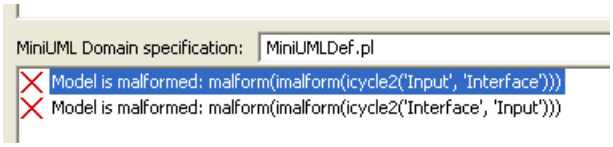


Fig. 12. Results of well-formedness proof after inheritance cycle is added to DSP metamodel.

can be constructed using GME. Figure 15 shows a DSP model created within GME using the generated modeling environment. As was the case with metamodeling, the extractor tool is attached to the generated modeling environment. This tool loads the formal definition of the domain into a Prolog engine and converts the domain model into terms, which are then checked for well-formedness. Figure 16 shows the result of activating the well-formedness checking tool on the DSP model of Figure 15. The tree-view labeled *Translation to Definite Clauses* shows the translation of each model element into a corresponding set of terms. The tree is organized according to the model hierarchy as represented by GME. Interestingly, the tool reports that the model is malformed. This occured because we augmented the DSP metamodel with an additional constraint that disallows a short-circuit of an input with an output. The connection from *In2* to *Out2* is such a short-circuit. This is made possible by directly annotating MiniMOF metamodels with Horn constraints. Figure 17 shows the annotated constraint in the DSP metamodel. After the domain generator converts the diagrammatic part of the metamodel into Horn clauses, it adds any annotations to the domain definition. Our approach combines the advantages of metamodeling with constraint languages in a consistent

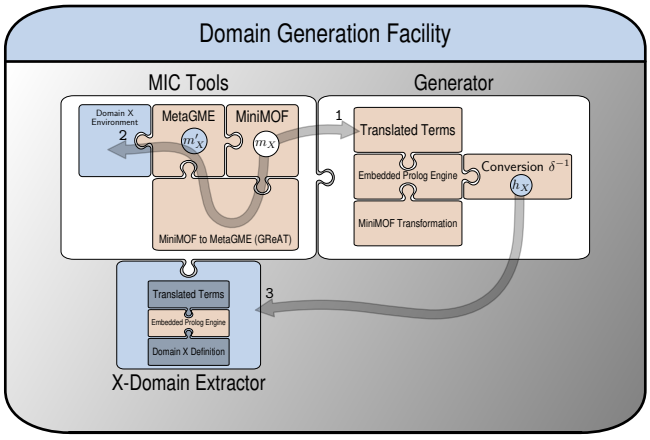


Fig. 13. Implementation of the domain generation facility using MetaGME, GME, GReAT, and an embedded Prolog engine.

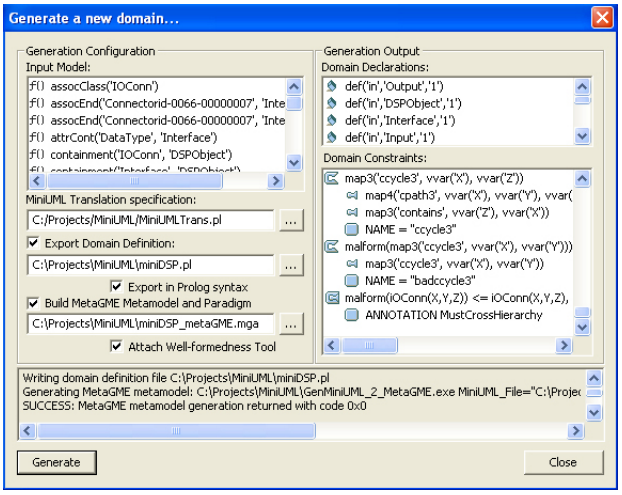


Fig. 14. Invocation of the domain generator component.

fashion. All of the features get translated into the same underlying formalism. In summary, MiniMeta is able to capture many of the core features of a mature DSML-based modeling framework in a mathematically precise manner. Additionally, MiniMeta can be connected to existing tools facilitating its immediate application.

V. Discussion and Conclusion

In this paper we explored the structural foundations of domain-specific modeling languages. We developed a generic structural semantics that addresses structural domain constraints, model transformations, and meta-modeling in a consistent fashion. We also applied our approach to a tried-and-tested DSML-based modeling framework called MIC, and this resulted in the MiniMeta framework. We showed that our structural semantics can capture the essential functionality of these tools, but do so in a mathematically precise manner. This opens

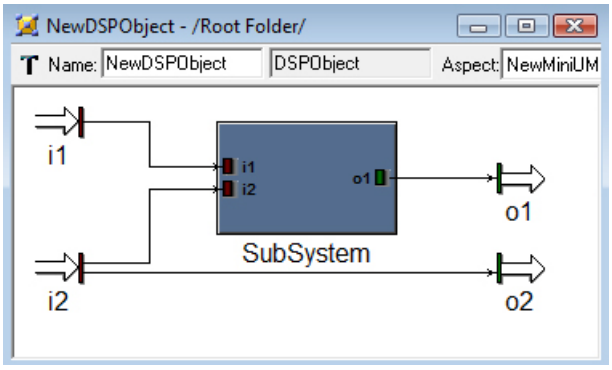


Fig. 15. An example DSP model created within GME

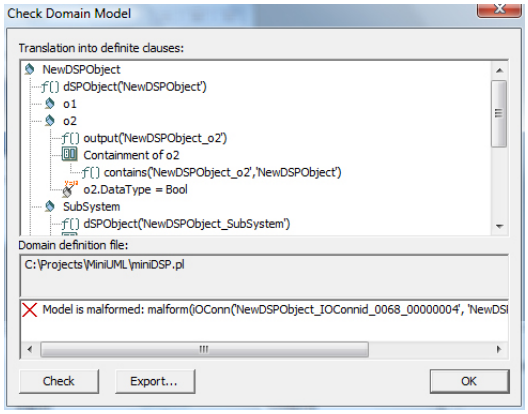


Fig. 16. Results of checking the DSP model against the formal domain definition

the door for formal analysis; several examples of which where shown in the paper. Beyond this, we implemented MiniMeta using MIC, and in doing so combine the advantages of each.

Our work has some interesting implications on current model-based tool suites. It is well-known that the basic concepts (classes, associations, etc...) in metamodeling languages are not sufficient to encode more complex structural constraints. The typical solution used by meta-modeling tools is to annotate metamodels with a constraint language like OCL (Object Constraint Language). However, this is often done without a precise notion of composition between metamodeling and constraints. With our approach, metamodels are translated into constraints, so additional constraints can be injected directly into the resulting interpretation of the metamodel. This provides a consistent and compositional view of metamodels and constraint annotations: They are just two different ways of describing domain constraints. Additionally, Horn logic extended with negation seems suitable for this purpose in that it is both expressive and there exists a large body of work in constructive theorem proving for this style of logic.

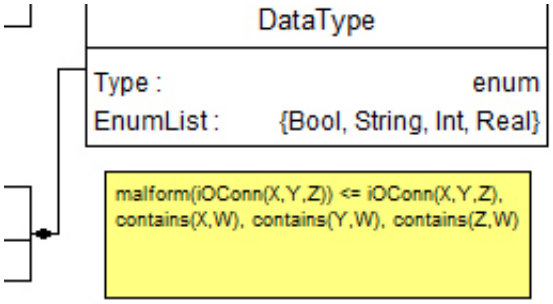


Fig. 17. Annotation of the DSP metamodel with an additional constraint

Existing approaches to metamodeling have taken an "instance semantics" perspective, wherein the elements of a DSML are "instances" of objects in the metamodel. This view binds all DSMLs to the particular presentation of the metamodeling language, making it difficult to achieve tool interoperability and evolution. Our approach eliminates this binding by building metamodeling from domains and transformations. A domain is a stand-alone mathematical entity that characterizes all the well-formed and malformed model realizations belonging to that domain. A metamodeling language generates domains, but after this generation process the domains are self-sufficient. Our tools also reflect this generality – The definitions of the MiniMeta domain and the transformation process are lists of axioms. These axioms can be changed at anytime without affecting the implementation of MiniMeta within MIC, and without affecting any legacy domains. This approach can be used to implement DSML-based modeling tools that do not rely on hard-coded metamodeling languages.

VI. Acknowledgments

Special thanks to Dr. Constantine Tsinakis. His expertise in Universal Algebra and his clarity of explanation helped to improve the algebraic presentation of this work. Also, special thanks to all the reviewers whose careful inspection lead to many improvements. This work was supported in part by the National Science Foundation (NSF award #CCR-0225610 and #CCF-0424422), the Air Force Office of Scientific Research (AFOSR award #FA9550-06-0312).

Appendix I
Partial MiniMOF Transformation

The axioms below calculate which types can contain other types, including containment capabilities endowed by inheritance edges. These axioms handle containment of both classes and association classes.

$$axiom(containment(x, ipath(h, y)), map_3(\texttt{cancontain}, var(\texttt{x}), var(\texttt{y}))) \leftarrow containment(x, y), ipath(h, y).$$

$$tail(containment(x, ipath(h, y)), map_2(x, var(\texttt{x}))) \leftarrow containment(x, y), class(x), ipath(h, y).$$

$$tail(containment(x, ipath(h, y)), map_4(x, var(\texttt{x}), var(\texttt{z}), var(\texttt{w}))) \leftarrow containment(x, y), assocClass(x), ipath(h, y).$$

$$tail(containment(x, ipath(h, y)), map_2(h, var(\texttt{y}))) \leftarrow containment(x, y), class(h), ipath(h, y).$$

These axioms are similar to the inheritance cycle equations of the MiniMOF domain. They are added to the domain description to prevent containment cycles.

$$axiom(\texttt{cloop}, malform(map_3(\texttt{contains}, var(\texttt{x}), var(\texttt{x})))) \leftarrow \texttt{true}.$$

$$tail(\texttt{cloop}, map_3(\texttt{contains}, var(\texttt{x}), var(\texttt{x}))) \leftarrow \texttt{true}.$$

$$axiom(\texttt{ccycle2}, malform(map_3(\texttt{contains}, var(\texttt{x}), var(\texttt{y})))) \leftarrow \texttt{true}.$$

$$tail(\texttt{ccycle2}, map_3(\texttt{contains}, var(\texttt{x}), var(\texttt{y}))) \leftarrow \texttt{true}.$$

$$tail(\texttt{ccycle2}, map_3(\texttt{contains}, var(\texttt{y}), vvar(\texttt{x}))) \leftarrow \texttt{true}.$$

The entire MiniMOF transformation is specified with about 200 transformation axioms. These axioms handle all the ways that containment, associations, attributes, and inheritance interact.

Appendix II
Example of Generated Domain in Prolog

```
%% Signature symbols
:- dynamic
   output/1,     dSPObject/1,
   interface/1,  input/1,
   malform/1,    enumvalue/2,
   dataType/2,   contains/2,
   ccyle3/2,     cancontain/2,
   iOConn/3,     cpath3/3,
   canconn/3.
%% Enumeration attribute values
enumvalue('DataType','Bool').
enumvalue('DataType','String').
enumvalue('DataType','Int').
enumvalue('DataType','Real').
```

```
%% Domain constraints
%% Attributes
malform(dataType(Y,V)) :- dataType(Y,V),
   dataType(Y,W), (V \== W).
malform(dataType(Y,V)) :- dataType(Y,V),
   \+interface(Y), \+input(Y),
   \+output(Y).
malform(iOConn(N,X,Y)) :- iOConn(N,X,Y),
   \+canconn('IOConn',X,Y).
%% Connection rules
canconn('IOConn',X,Y) :- interface(X),
   interface(Y).
canconn('IOConn',X,Y) :- input(X),
   interface(Y).
canconn('IOConn',X,Y) :- output(X),
   interface(Y).
canconn('IOConn',X,Y) :- interface(X),
   input(Y).
canconn('IOConn',X,Y) :- interface(X),
   output(Y).
canconn('IOConn',X,Y) :- input(X),
   input(Y).
canconn('IOConn',X,Y) :- input(X),
   output(Y).
canconn('IOConn',X,Y) :- output(X),
   input(Y).
canconn('IOConn',X,Y) :- output(X),
   output(Y).
%% Containment rules
malform(contains(X,Y)) :- contains(X,Y),
   \+cancontain(X,Y).
cancontain(X,Y) :- iOConn(X,Z,W),
   dSPObject(Y).
cancontain(X,Y) :- interface(X),
   dSPObject(Y).
cancontain(X,Y) :- dSPObject(X),
   dSPObject(Y).
cancontain(X,Y) :- input(X),
   dSPObject(Y).
cancontain(X,Y) :- output(X),
   dSPObject(Y).
malform(dataType(Y,Z)) :- dataType(Y,Z),
   \+enumvalue('DataType',Z).
malform(contains(X,X)) :- contains(X,X).
malform(contains(X,Y)) :- contains(X,Y),
   contains(Y,X).
cpath3(X,Y,Z) :- contains(X,Y),
   contains(Y,Z), (X \== Y),
   (X \== Z), (Y \== Z).
ccycle3(X,Z) :- cpath3(X,Y,Z),
   contains(Z,X).
malform(ccycle3(X,Y)) :- ccycle3(X,Y).
%% Additional annotations
malform(iOConn(X,Y,Z)) :- iOConn(X,Y,Z),
   contains(X,W), contains(Y,W),
   contains(Z,W).
```

APPENDIX III
MINIMOF DOMAIN

Table V describes the Horn domain for representing domains with constraints in Horn logic. Please see tables VI and VII on the following pages for a description of the function symbols and constraint axioms of the MiniMOF domain.

The Horn Domain

Define. $def(x, y, z)$ defines a function symbol y with arity z in a signature x. If the same symbol appears in multiple signatures, then the arity of the symbol must be the same in every signature. There may be two signatures, one for Υ (`sig`) and one for Υ_C (`con`).

$\{(def, 3), (sigtype, 1)\} \subset \Upsilon^{\mathcal{H}}$
$\mathbf{restrict}(sigtype, \{sigtype(\texttt{sig}), sigtype(\texttt{con})\})$
$malform(def(x, y, z)) \leftarrow def(x, y, z) \wedge def(x', y, z'), (z \neq z')$
$malform(x, y, z) \leftarrow def(x, y, z), \neg sigtype(x)$

The inverse representation function δ^{-1} of a term $def(x, y, z)$ yields a symbol definition of the form $(y, z) \in \Upsilon_x$.

Map$_n$. $map_n(x, y_1, y_2, \ldots, y_{n-1})$ converts an n-ary term to *prenix form*. The symbol name x must be defined with a def. The domain definition provides a finite number k of map symbols.

$\{(map_2, 2), \ldots, (map_k, k)\} \subset \Upsilon^{\mathcal{H}}$
$malform(map_2(x, y_1)) \leftarrow map_2(x, y_1), \neg def(t, x, z)$
...
$malform(map_k(x, y_1, \ldots, y_{k-1})) \leftarrow map_k(x, y_1, \ldots, y_{k-1}), \neg def(t, x, z)$

The inverse representation function δ^{-1} on $map_n(x, y_1, \ldots, y_{n-1})$ yields a literal of the form $\delta_f^{-1}(x)(\delta^{-1}(Y_1), \ldots, \delta^{-1}(y_1))$.

Axiom/Tail. $axiom$ defines the head of an axiom and assigns it a unique identifier. $tail$ adds a tail literal to an axiom by referring to the axiom's unique identifier. Each $tail$ must be added to an axiom that has been defined with $axiom$. Every axiom identifier must be unique.

$\{(axiom, 2), (tail, 2)\} \in \Upsilon^{\mathcal{H}}$
$malform(tail(x, y)) \leftarrow tail(x, y), \neg axiom(x, z)$
$malform(axiom(x, y)) \leftarrow axiom(x, y), axiom(x, z), (y \neq z)$

Given $axiom(x, h)$ and tails $tail(x, t_1), \ldots, tail(x, t_m)$, the inverse representation function δ^{-1} yields a clause with the corresponding head and all tails conjuncted together: $\delta^{-1}(h) \leftarrow \delta^{-1}(t_1), \ldots, \delta^{-1}(t_m)$.

Neg/Neq/Var. $neg(x)$ indicates the negation of literal x. $neq(x, y)$ indicates the disequality $x \neq y$. $var(x)$ indicates that x is a variable.

$\{(neg, 1), (neq, 2), (var, 1)\} \subset \Upsilon^{\mathcal{H}}$

The inverse representation function δ^{-1} of $neg(x)$ yields the negated term $\neg\delta^{-1}(x)$ and $neq(x, y)$ yields the disequality $\delta^{-1}(x) \neq \delta^{-1}(y)$. Finally, $\delta^{-1}(var(x))$ yields a variable $\delta_v^{-1}(x)$.

TABLE V
ENCODING OF CONCEPTS IN THE HORN DOMAIN.

Encoding for Vertex Symbols

Class. $class(x)$ denotes a class named x.

$(class, 1) \in \Upsilon$

Association Class. $assocClass(x)$ denotes an association class named x.

$(assocClass, 1) \in \Upsilon$

Attribute Class. $attribute(x, y)$ denotes an attribute named x of type y. $enum(x, y)$ indicates that attribute x can take the enumerated value y.

$\{(attribute, 2), (enum, 2)\} \subset \Upsilon$
$\mathbf{restrict}(type, \{type(\texttt{bool}), type(\texttt{string}), type(\texttt{enum})\})$

An attribute must have a proper type. Also, an attribute cannot have an enum list, unless it is an enum attribute.

$malform(attribute(x, y)) \leftarrow attribute(x, y), \neg type(y)$
$malform(enum(x, y)) \leftarrow enum(x, y), \neg attribute(x, z)$
$malform(enum(x, y)) \leftarrow enum(x, y), attribute(x, z), (z \neq \texttt{enum})$

Connector. $connector(x)$ denotes a connector named x. A connector is good ($conngood(x)$) if it has the appropriate edges, and every connector must be good.

$(connector, 1) \in \Upsilon,\ (conngood, 1) \in \Upsilon_C$
$conngood(w) \leftarrow \begin{pmatrix} connector(x), assocEnd(x, y, \texttt{src}), \\ assocEnd(x, z, \texttt{dst}), association(x, w) \end{pmatrix}$
$malform(connector(x)) \leftarrow connector(x), \neg conngood(x)$

TABLE VI
ENCODING OF MINIMOF VERTEX PRIMITIVES

Encoding for Edge Symbols

Containment. $containment(y, x)$ denotes the containment relationship of y in x. A Containment edge must terminate on a *Class*. It may begin on another *Class* or *Association Class*.

$(containment, 2) \in \Upsilon$
$malform(containment(y, x)) \leftarrow containment(y, x), \neg class(x)$
$malform(containment(y, x)) \leftarrow containment(y, x), \neg class(y), \neg assocClass(y)$

Attribute Containment. $attrCont(y, x)$ indicates an attribute containment relationship of y in x. An attribute containment edge must begin on an *Attribute*. It may terminate on a *Class* or *Association Class*.

$(attrCont, 2) \in \Upsilon$
$malform(attrCont(y, x)) \leftarrow attrCont(y, x), \neg attribute(y)$
$malform(attrCont(y, x)) \leftarrow attrCont(y, x), \neg class(x), \neg assocClass(x)$

Association. $association(x, y)$ denotes an association relationship from *Connector* x to *Assocation Class* y. This is the only relationship allowed.

$(association, 2) \in \Upsilon$
$malform(association(x, y)) \leftarrow association(x, y), \neg connector(x)$
$malform(association(x, y)) \leftarrow association(x, y), \neg assocClass(y)$

Association Endpoint. $assocEnd(x, y, z)$ indicates an association endpoint relation from *Connector* x to *Class* y with incidence z. z must a member of the closed unary relation $incidence$.

$\{(assocEnd, 3), (incidence, 1)\} \subset \Upsilon$
restrict$(incidence, \{incidence(\texttt{src}), incidence(\texttt{dst})\})$
$malform(assocEnd(x, y, z)) \leftarrow assocEnd(x, y, z), \neg connector(x)$
$malform(assocEnd(x, y, z)) \leftarrow assocEnd(x, y, z), \neg class(y)$
$malform(assocEnd(x, y, z)) \leftarrow assocEnd(x, y, z), \neg incidence(z)$

Inheritance. $inheritance(y, x)$ indicates the inheritance relationship y inherits from x. An inheritance relationship can start and end on a *Class* or *Association Class*. There should be no directed cycles consisting only of inheritance edges.

$(inheritance, 2) \in \Upsilon,\ (imalform, 1) \in \Upsilon_C$
$malform(inheritance(y, x)) \leftarrow inheritance(x, y), \neg class(x), \neg assocClass(x)$
$malform(inheritance(y, x)) \leftarrow inheritance(x, y), \neg class(y), \neg assocClass(y)$
$malform(imalform(x)) \leftarrow imalform(x)$

See Section IV-B for the definition of axioms concerning $imalform$.

TABLE VII
ENCODING OF MINIMOF EDGE PRIMITIVES

REFERENCES

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] AÏT-KACI, H. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[3] ALUR, R., ETESSAMI, K., AND YANNAKAKIS, M. Inference of message sequence charts. In *Proceedings of the International Conference on Software Engineering (ICSE 00)* (2000), pp. 304–313.

[4] ANDREAE, C., NOBLE, J., MARKSTRUM, S., AND MILLSTEIN, T. A framework for implementing pluggable type systems. *SIGPLAN Not. 41*, 10 (2006), 57–74.

[5] ATKINSON, C., AND KÜHNE, T. Model-driven development: A metamodeling foundation. *IEEE Software 20*, 5 (2003), 36–41.

[6] BANACH, R., AND FRASER, S. Retrenchment and the b-toolkit. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users* (april 2005), pp. 203–221.

[7] BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., GUERNIC, P. L., AND DE SIMONE, R. The synchronous languages twelve years later. *Proceedings of the IEEE 91*, 1 (2003), 64–83.

[8] BÉZIVIN, J., AND GERBÉ, O. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th Conference on Automated Software Engineering (ASE 01)* (2001), pp. 273–280.

[9] BÖRGER, E. Unsolvable decision problems for prolog programs. In *Computation theory and logic* (London, UK, 1987), Springer-Verlag, pp. 37–48.

[10] BRUCKER, A. D., RITTINGER, F., AND WOLFF, B. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science 9*, 2 (Feb. 2003), 152–172.

[11] BURRIS, S. N., AND SANKAPPANAVAR, H. P. *A Course in Universal Algebra*. Springer-Verlag, 1981.

[12] CARNES, J. R., MISRA, A., AND SZTIPANOVITS, J. Model-integrated toolset for fault detection, isolation and recovery (fdir). In *ECBS '96: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems* (Washington, DC, USA, 1996), IEEE Computer Society, p. 356.

[13] CHAN, D. An extension of constructive negation and its application in coroutining. *In Proceedings of NACLP, The MIT Press* (1989), 447–493.

[14] CHEN, K., SZTIPANOVITS, J., NEEMA, S., EMERSON, M., AND ABDELWAHED, S. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)* (September 2005).

[15] CHOMSKY, N. On certain formal properties of grammars. *Information and Control 2*, 2 (1959), 137–167.

[16] COLMERAUER, A., AND ROUSSEL, P. The birth of prolog. In *HOPL Preprints* (1993), pp. 37–52.

[17] CSERTÁN, G., HUSZERL, G., MAJZIK, I., PAP, Z., PATARICZA, A., AND VARRÓ, D. Viatra - visual automated transformations for formal verification and validation of uml models. In *ASE* (2002), pp. 267–270.

[18] CZARNECKI, K., AND HELSEN, S. Feature-based survey of model transformation approaches. *IBM Syst. J. 45*, 3 (2006), 621–645.

[19] DANTSIN, E., EITER, T., GOTTLOB, G., AND VORONKOV, A. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33*, 3 (2001), 374–425.

[20] DE ALFARO, L., AND HENZINGER, T. A. Interface-based design. In *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School, Kluwer* (2004), Kluwer.

[21] DREWES, F., HOFFMANN, B., AND PLUMP, D. Hierarchical graph transformation. *J. Comput. Syst. Sci. 64*, 2 (2002), 249–283.

[22] EHRIG, H., EHRIG, K., PRANGE, U., AND TAENTZER, G. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inform. 74*, 1 (2006), 31–61.

[23] EHRIG, H., PRANGE, U., AND TAENTZER, G. Fundamental theory for typed attributed graph transformation. In *ICGT* (2004), pp. 161–177.

[24] EMERSON, M., SZTIPANOVITS, J., AND BAPTY, T. A mof-based metamodeling environment. *Journal of Universal Computer Science 10*, 10 (October 2004), 1357–1382.

[25] EMERSON, M. J., SZTIPANOVITS, J., AND BAPTY, T. A mof-based metamodeling environment. *J. UCS 10*, 10 (2004), 1357–1382.

[26] EVANS, A., FRANCE, R. B., AND GRANT, E. S. Towards formal reasoning with uml models. In *In Proceedings of the Eighth OOPSLA Workshop on Behavioral Semantics*.

[27] FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. Composable attribute grammars: support for modularity in translator design and implementation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1992), pp. 223–234.

[28] FOSDICK, L. D., AND OSTERWEIL, L. J. Data flow analysis in software reliability. *ACM Comput. Surv. 8*, 3 (1976), 305–330.

[29] GABOR KARSAI, ADI AGRAWAL, F. S. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science 9*, 11 (November 2003), 1296–1321.

[30] GELFOND, M., AND LIFSCHITZ, V. The stable model semantics for logic programming. In *ICLP/SLP* (1988), pp. 1070–1080.

[31] GENERIC MODELING FRAMEWORK TEAM. GMF Documentation, http://wiki.eclipse.org/, 2007.

[32] GIRSCHICK, M., KÜHNE, T., AND KLAR, F. Generating systems from multiple levels of abstraction. In *Proceedings of the Trends in Enterprise Application Architecture (TEAA 2006)* (2006), pp. 127–141.

[33] GÖSSLER, G., GRAF, S., MAJSTER-CEDERBAUM, M. E., MARTENS, M., AND SIFAKIS, J. Ensuring properties of interaction systems. In *Program Analysis and Compilation* (2006), pp. 201–224.

[34] GRAF, S., OBER, I., AND OBER, I. A real-time profile for uml. *STTT 8*, 2 (2006), 113–127.

[35] GRAY, J. G., AND ROYCHOUDHURY, S. A technique for constructing aspect weavers using a program transformation engine. In *AOSD* (2004), pp. 36–45.

[36] GREENFIELD, J. Using domain-specific languages, patterns, frameworks, and tools to assemble applications. In *SPLC* (2004), p. 324.

[37] HAMON, G., AND RUSHBY, J. An operational semantics for stateflow. *Proceedings of the conference on Fundamental Approaches to Software Engineering (FASE'04) Volume 2984 of LNCS* (2005), 229–243.

[38] HAREL, D., AND KUGLER, H. The rhapsody semantics of statecharts (or, on the executable core of the uml) - preliminary version. In *SoftSpez Final Report* (2004), pp. 325–354.

[39] HAREL, D., KUGLER, H., AND PNUELI, A. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling* (2005), pp. 309–324.

[40] HAREL, D., AND NAAMAD, A. The statemate semantics of statecharts. *ACM Transactions Software Engineering Methodologies 5*, 4 (1996), 293–333.

[41] HAREL, D., AND RUMPE, B. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer 37*, 10 (2004), 64–72.

[42] HENZINGER, T. A. The theory of hybrid automata. In *Proceedings of the International Conference on Logic in Computer Science (LICS 96)* (1996), pp. 278–292.

[43] HENZINGER, T. A., KIRSCH, C. M., SANVIDO, M. A., AND PREE, W. From control models to real-time code using giotto. *Control Systems Magazine 2*, 1 (2003), 50–64.

[44] INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS. GME 5 User's Guide, http://www.isis.vanderbilt.edu/, 2005.

[45] J. BURCH, R. PASSERONE, A. S.-V. Modeling techniques in design-by-refinement methodologies. In *Integrated Design and Process Technology* (June 2002).

[46] JACKSON, D. A comparison of object modelling notations: Alloy, uml and z. Tech. rep., August 1999.

[47] JACKSON, D. Automating first-order relational logic. In *SIGSOFT FSE* (2000), pp. 130–139.

[48] JACKSON, E., SCHULTE, W., AND SZTIPANOVITS, J. The power of rich syntax for model-based deveopment. Tech. Rep. MSR-TR-2008-86, ftp://ftp.research.microsoft.com/pub/tr/TR-2008-86.pdf, 2008.

[49] JACKSON, E., AND SZTIPANOVITS, J. Constructive techniques for meta- and model-level reasoning. In *Proceedings of MODELS'07*.

[50] JACKSON, E. K., AND SCHULTE, W. Compositional modeling for data-centric business applications. In *Proceedings of the Workshop on Software Engineering (SC 2008)* (2008).

[51] JACKSON, E. K., AND SCHULTE, W. Model generation for horn logic with stratified negation. In *Proceedings of the IFIP WG 6.1 International Conference Formal Techniques for Networked and Distributed Systems (FORTE 2008)* (2008), pp. 1–20.

[52] JACKSON, E. K., AND SZTIPANOVITS, J. Correct-ed through construction: A model-based approach to embedded systems reality. In *ECBS* (2006), pp. 164–176.

[53] JACKSON, E. K., AND SZTIPANOVITS, J. Constructive techniques for meta- and model-level reasoning. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)* (2007), pp. 405–419.

[54] JOUAULT, F., AND BÉZIVIN, J. Km3: A dsl for metamodel specification. In *FMOODS* (2006), pp. 171–185.

[55] JÜRJENS, J. Sound methods and effective tools for model-based security engineering with uml. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 322–331.

[56] KAHN, G., AND MACQUEEN, D. B. Coroutines and networks of parallel processes. In *IFIP Congress* (1977), pp. 993–998.

[57] KAKITA, S., WATANABE, Y., DENSMORE, D., DAVARE, A., AND SANGIOVANNI-VINCENTELLI, A. L. Functional model exploration for multimedia applications via algebraic operators. In *ACSD* (2006), pp. 229–238.

[58] KARSAI, G., SZTIPANOVITS, J., LEDECZI, A., AND BAPTY, T. Model-integrated development of embedded software. *Proceedings of the IEEE 91*, 1 (January 2003), 145–164.

[59] KNUTH, D. E. The genesis of attribute grammars. In *In Proceedings of Attribute Grammars and their Applications* (1990), pp. 1–12.

[60] KÖNIGS, A., AND SCHÜRR, A. Multi-domain integration with mof and extended triple graph grammars. In *Language Engineering for Model-Driven Software Development* (2004).

[61] KÜHNE, T. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM) 5*, 4 (December 2006), 369–385.

[62] LARSEN, K. G., AND NIEBERT, P., Eds. *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers* (2003), vol. 2791 of *Lecture Notes in Computer Science*, Springer.

[63] LAWVERE, W. F., AND SCHANUEL, S. H. *Conceptual Mathematics : A First Introduction to Categories*. Cambridge University Press, October 1997.

[64] LEDECZI, A., MAROTI, M., BAKAY, A., KARSAI, G., GARRETT, J., THOMASON, C., NORDSTROM, G., SPRINKLE, J., AND VOLGYESI, P. The generic modeling environment. *Workshop on Intelligent Signal Processing* (May 2001).

[65] LEE, E. A., AND NEUENDORFFER, S. Actor-oriented models for codesign: Balancing re-use and performance. *Formal Methods and Models for Systems, Kluwer* (2004).

[66] LEE, E. A., NEUENDORFFER, S., AND WIRTHLIN, M. J. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers 12*, 3 (2003), 231–260.

[67] LEE, E. A., AND PARKS, T. M. Dataflow process networks. *Proceedings of the IEEE* (may 1995), 773–799.

[68] LONG, E., MISRA, A., AND SZTIPANOVITS, J. Increasing productivity at saturn. *IEEE Computer 31*, 8 (1998), 35–43.

[69] MARCANO, R., AND LEVY, N. 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.

[70] MENS, T., GORP, P. V., VARRÓ, D., AND KARSAI, G. Applying a model transformation taxonomy to graph transformation technology. *Electr. Notes Theor. Comput. Sci. 152* (2006), 143–159.

[71] MINKER, J. Logic and databases: A 20 year retrospective. In *Logic in Databases* (1996), pp. 3–57.

[72] MULLER, P.-A., FLEUREY, F., AND JÉZÉQUEL, J.-M. Weaving executability into object-oriented meta-languages. In *MoDELS* (2005), pp. 264–278.

[73] NEEMA, S., KALMAR, Z., SHI, F., VIZHANYO, A., AND KARSAI, G. A visually-specified code generator for simulink/stateflow. In *VL/HCC* (2005), pp. 275–277.

[74] NEEMA, S., SZTIPANOVITS, J., KARSAI, G., AND BUTTS, K. Constraint-based design-space exploration and model synthesis. In *EMSOFT* (2003), pp. 290–305.

[75] OBJECT MANAGEMENT GROUP. Meta Object Facility Specification v1.4, http://www.omg.org/docs/formal/02-04-03.pdf, 2002.

[76] OBJECT MANAGEMENT GROUP. MDA Guide version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[77] OBJECT MANAGEMENT GROUP. Meta Object Facility (MOF) 2.0 Core Specification, http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[78] OBJECT MANAGEMENT GROUP. Unified Modeling Language: Infrastructure version 2.0, 3rd revised submission to OMG RFP, http://www.omg.org/docs/ad/00-09-02.pdf, 2003.

[79] OBJECT MANAGEMENT GROUP. Object Constraint Language v2.0, http://www.omg.org/docs/formal/06-05-01.pdf, 2006.

[80] OBJECT MANAGEMENT GROUP. Unified Modeling Language: Superstructure version 2.1.1, http://www.omg.org/docs/formal/07-02-06.pdf, 2007.

[81] OREJAS, F., EHRIG, H., AND PRANGE, U. A logic of graph constraints. In *FASE* (2008), pp. 179–198.

[82] PAAKKI, J. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv. 27*, 2 (1995), 196–255.

[83] PINTO, A., CARLONI, L. P., PASSERONE, R., AND SANGIOVANNI-VINCENTELLI, A. L. Interchange format for hybrid systems: Abstract semantics. In *HSCC* (2006), pp. 491–506.

[84] PRZYMUSINSKI, T. C. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1989), ACM, pp. 11–21.

[85] SANGIOVANNI-VINCENTELLI, A., CARLONI, L., BERNARDINIS, F. D., AND SGROI, M. Benefits and challenges for platform-based design. In *Proceedings of the Design Automation Conference (DAC'04)* (June 2004).

[86] TORLAK, E., AND JACKSON, D. Kodkod: A relational model finder. In *In Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)* (2007), pp. 632–647.

[87] VARRÓ, D., AND PATARICZA, A. Vpm: A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml (the mathematics of metamodeling is metamodeling mathematics). *Software and System Modeling 2*, 3 (2003), 187–210.

[88] WEIJLAND, W. P. Semantics for logic programs without occur check. In *ICALP* (1988), pp. 710–726.