Towards A Formal Foundation For Domain Specific Modeling Languages [*]

Ethan K. Jackson
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
ejackson@isis.vanderbilt.edu

Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
janos.sztipanovits@vanderbilt.edu

ABSTRACT

Embedded system design is inherently domain specific and typically model driven. As a result, design methodologies like OMG's model driven architecture (MDA) and model integrated computing (MIC) evolved to support domain specific modeling languages (DSMLs). The success of the DSML approach has encouraged work on the heterogeneous composition of DSMLs, model transformations between DSMLs, approximations of formal properties within DSMLs, and reuse of DSML semantics. However, in the effort to produce a mature design approach that can handle both the structural and behavioral semantics of embedded system design, many foundational issues concerning DSMLs have been overlooked. In this paper we present a formal foundation for DSMLs and for their construction within metamodeling frameworks. This foundation allows us to algorithmically decide if two DSMLs or metamodels are equivalent, if model transformations preserve properties, and if metamodeling frameworks have meta-metamodels. These results are key to building correct embedded systems with DSMLs.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.1 [**Software**]: Programming Languages—*Formal Definitions and Theory*

General Terms

Design, Languages, Theory

Keywords

Embedded Systems, Semantics, Metamodeling, Formal Logic, Horn Logic

[*]This research was supported by NSF grant CCR-0225610 and CCF-0424422.

1. INTRODUCTION

Embedded systems are application specific, and this affects the embedded system design process at a fundamental level. Embedded hardware must be designed to meet the constraints imposed by the physicality of the application (power, size, weight, etc...), and embedded software must be written to respect assumptions about the application environment. This pressure to design for a specific application results in brittle designs that are not easily ported to other application contexts. Researchers recognized this problem and proposed a solution: Define the application context (i.e. constraints, communication mechanisms, and time models) before implementing the application.

Domain specific modeling languages (DSMLs) were proposed for specifying the application contexts of embedded systems [7]. A DSML encapsulates a context, also called a *domain*, by providing:

1. A set of components or constructs with which embedded hardware/software can be modeled.

2. A set of constraints that enforce proper use of components.

3. A set of semantic mappings that generate simulation traces, embedded code, and verifications results from models.

A sizable repository of tools support the construction and utilization of DSMLs. For example, the modeling tool GME allows users to construct models that belong to a particular domain [2]. It also enforces domain constraints at modeling time, and we previously showed how this helps to design correct embedded systems [14]. The tool *MetaGME* [13] allows users to quickly specify new domains. The model transformation tool *GReAT* [8] allows models from one domain to be transformed into models in another domain. Finally, the previous work on *semantic anchoring* allows semantic mappings to be formally defined [15].

Though the development of DSML-based embedded systems has been quite successful, we still have little formal understanding of DSMLs. This means, for example, that we cannot tell if two domains are the same. We do not know if model transformations preserve properties (deadlock freedom, determinism). Without a formal understanding, our designs remain inextricably tied to the tools with which they were built. However, if DSMLs can be formalized, then their meanings becomes precise and independent of specific tools.

In this paper we present a formalization of DSMLs that addresses these issues. Section 2 provides an abstract mathematical framework for formalizing DSMLs. In Section 3 we describe a restricted, yet powerful concretization based on Horn logic. We also discuss techniques for analysis and verification of DSML properties. We conclude in Section 4 by discussing how our formalism impacts current and future design tools.

2. ABSTRACT MODEL

2.1 Domains And Structural Semantics

Every application context is expressed by a *domain*, which is the set of all *structurally well-formed models* for that context. A common domain is shown in the the foreground of Figure 1. The basic building blocks of this domain are the the hardware components (ASICs and cards) that can be plugged into the circuit board. Each block on 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 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* is a description that has no remaining degrees of freedom, e.g., every place on the circuit board has some hardware assigned to it. A *well-formed model* is a model that satisfies all the constraints imposed on its construction. The set of all well-formed models contains all the meaningful structures of a domain. It is important to note that the set of well-formed models can be defined without giving a meaning to the constructs that participate in the model. For example, we do not need to give any details about what CPUs, RAMs, and Buses do in order to check well-formedness of a circuit board.

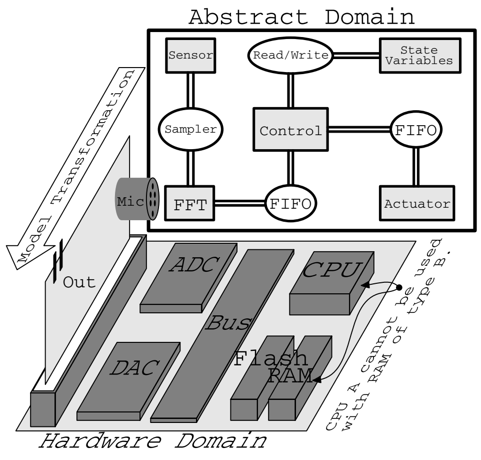

Figure 1: Example of two domains.

The set of well-formed models defines the *structural semantics* of domain. Another way to phrase this is to say that the structural semantics of a domain is a decision procedure for checking model well-formedness[1]. The structural semantics serves the single purpose of identifying the important models within a larger set of models. Later on we will attach more precise meanings to well-formed models; these other meanings are also called "semantics". A domain and its structural semantics are given by:

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

The model realizations in R_Υ are all the ways that models can be built from the available primitives. The set of *well-formed models* in a domain is the set of all models that satisfy the constraints. We write this set as

$$D(\Upsilon, C) = \{r \in R_\Upsilon \mid r \models C\}. \tag{1}$$

The notation $r \models C$ can be read as *"r satisfies the constraints C"*.

DSML tools make significant use of the structural semantics, so it must be formalized carefully. The first essential issue is how the set of all model realizations R_Υ relates to the set of concepts Υ. The most obvious formalization is to let Υ be a set of sets, where each set enumerates all objects of a particular type. This approach is analogous to *multi-sorted algebra* [18] where the signature of the algebra is given by a set of types A, called the *index set*, and a collection of sets $(\mathbf{A}_i)_{i\in A}$ that are called the *carriers of the algebra*. Each carrier set $\mathbf{A}_i$ contains all the objects of type $i \in A$. A model is some subset of objects taken from each carrier:

$$\begin{array}{c} \Upsilon = \langle A,\ (\mathbf{A}_i)_{i\in A}\rangle \\ R_\Upsilon = \times_{i\in A}\ (\mathcal{P}(\mathbf{A}_i)) \end{array} \tag{2}$$

Though reasonable, this approach makes it difficult to define objects that are relations on other objects. For example, many domains are graph-based wherein some objects are treated as vertices and others as edges. Typically, an edge set E is a binary relation on vertices, i.e. $E_{i,j} \subseteq \mathbf{A}_i \times \mathbf{A}_j$. In the multi-sorted algebra approach a relation is defined on top of the carrier sets, and this makes relations less basic than non-relations. The matter is further complicated by objects that may be relations of objects that are themselves relations. For example, edges and vertices may be "contained" inside of another object by a *containment relation*. In general, we must support arbitrary n-ary relations over objects that may themselves be relations.

Formal logic provides a natural way to specify relations and gives a well-known construct to generate all possible uses of these relations, which are the models. However, before presenting the formal notation, we will give an example to build intuition about this construct. Figure 2 shows a model that belongs to a domain for Digital Signal Processing (DSP) systems. We will work backwards from this single model to the domain of all DSP models. To begin, we must extract the primitive concepts used to build DSP systems. Examining Figure 2, we see that this model 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). The zoomed in box shows that the primitives have interfaces, which are sets of uniquely identifiable *ports*. In order to capture these concepts, we will provide the names for a set of n-ary relations for encoding

[1]This assumes that well-formedness is decidable; we require this to be the case.

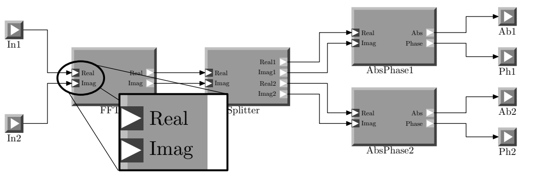

Figure 2: Detailed view of a DSP model.

the modeling concepts. The names are called *function symbols* in formal logic, and they form the set of concepts Υ. Importantly, Υ does not contain an enumeration of the relations; only the names of the relations. Below are the basic concepts of the DSP domain written as n-ary function symbols.

Primitives of DSP Domain

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

The function symbols clearly encode the concepts available to the DSP modeling language. These functions are defined over a set of object names and values; the places where the functions are defined gives information about the support. For example, Figure 2 shows that there is an `FFT` primitive in the DSP domain. We capture this by writing $prim(\texttt{FFT})$, where the constant `FFT` is a name or constant from some underlying support set. The fact that $prim(X)$ is defined at the constant `FFT` indicates that an `FFT` is a primitive. A model is just a listing of all the places where the functions are well-defined. In the terminology of formal logic, we consider a model to be a set of variable-free definite clauses. The table below shows a partial encoding of the DSP model as variable-free definite clauses[2].

Partial Encoding of Figure 2

Primitives	$prim(\texttt{FFT}),\ prim(\texttt{Splitter}),\ prim(\texttt{Phase})$
FFT 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	$flow(insig(\texttt{In1}), insig(\texttt{In1}),$ $inst(\texttt{FFT}, prim(\texttt{FFT})), iport(prim(\texttt{FFT}),\texttt{Real})),$ $\ldots$

Formally, function symbols stand for n-ary functions over the strings of some finite alphabet Σ. It is not necessary to explicitly define the functions; instead we assume they are one-to-one with disjoint codomains, and then indicate the points where the functions are defined. For example, $v(\texttt{A}), v(\texttt{B})$ indicates that v is defined at `A` and `B`, and $v(\texttt{A}) \neq v(\texttt{B})$. This definition also permits nesting of the symbols as in $e(v(\texttt{A}), v(\texttt{B}))$. When a symbol is used as a logical predicate, it returns **true** if its arguments are defined at the point of evaluation, and **false** otherwise. These are common assumptions made for function symbols in formal logic.

A model is encoded as a set of variable-free definite clauses, therefore the set of all models R_Υ contains all possible sets of variable-free definite clauses that can be formed from Υ and Σ^*. This set can be defined in terms of the well-known Herbrand Universe [11], which is a construction of all the variable-free definite clauses.

Definition 1. Given a set of n-ary function symbols Υ and a finite alphabet Σ, the *Herbrand Universe* $\mathcal{H}(\Upsilon,\Sigma)$ is defined with the following induction:

1. if $s \in \Sigma^*$, then s is in $\mathcal{H}$
2. if $f \in \Upsilon$ is an n-ary function symbol and $t_1,\ldots,t_n$ are in $\mathcal{H}$, then $f(t_1,\ldots,t_n) \in \mathcal{H}$.

The set of all models is the set of all subsets of $\mathcal{H}$, i.e., the powerset of $\mathcal{H}$, $\mathcal{P}(\mathcal{H})$.

Definition 2. Given a set of n-ary function symbols Υ and a finite alphabet Σ:

$$R_\Upsilon = \mathcal{P}\left(\ \mathcal{H}(\Upsilon,\Sigma)\ \right) \tag{3}$$

Notice that R_Υ contains the cross product style models that are found in the multi-sorted algebra approach, but it also contains all possible mixing of n-ary relations. These types of models are not basic for multi-sorted algebras. Gurevich et al [10] designed Abstract State Machines (ASM) around a similar use of function symbols. We have generalized this by using the Herbrand Universe, which we believe provides an elegant approach to the structural semantics of DSMLs.

Embedding the structural semantics inside of formal logic also provides a natural framework for specifying and evaluating well-formedness rules. A model realization r is a conjunction of variable-free definite clauses, i.e $r = \bigwedge t_i,\ t_i \in \mathcal{H}(\Sigma,\Upsilon)$. Therefore, we can compose a model with a set C of logical statements via conjunction, $r \wedge C$, and this yields another perfectly good set of logical statements. We will use this composition mechanism to deduce well-formedness of a model r. To do this, we construct a set of constraints C so that the composition $r \wedge C$ can be used to deductively prove well-formedness or malformedness of any model in R_Υ. Specifically, we assume that there is a special function symbol $wellform(\cdot) \notin \Upsilon$, such that a model r is well-formed if $wellform(X)$ can be proven for some value of X. We can then utilize the relevant inference procedure $\vdash$ to prove well-formedness. Not all logical styles are closed under negation, so we may have to define a domain in terms of malformedness. In these cases there is a symbol $malform(\cdot)$, and a model is malformed if $malform(X)$ can be proved true for some X. We call domains with constraints like the former *positive domains* and the latter *negative domains.*

Our approach highlights the second essential issue of structural semantics: How expressive should the structural semantics be? We can answer this question by observing that

[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.

the logic chosen for constraints also controls the expressiveness of the structural semantics through the relevant inference procedure. The table below lists multiple candidate logics and lists the properties of their inference procedures:

Candidate Logics

Logic	2^{nd} Order	1^{st} Order	Horn	Horn+NAF
Decidability	Undecid.	Semi.	Decid.	Decid.
Monotonicity	Yes	Yes	Yes	No
Inference Algo.	Varies	Unit	SLD	SLD
Complexity	Varies	NP	Poly	Poly
Soundness	Varies	Yes	Yes	No
Completeness	Varies	Yes	No	No

Expressive logics like first or second order logic yield domains with complex properties. However, the expressiveness of these logics can also cause well-formedness to become semidecidable or undecidable. We must therefore use a more restricted logic where decidability is guaranteed. In Section 3 we will concretize our abstract discussion by choosing a highly decidable subset of first order logic, called *Horn logic* [12]. Other work using Horn logic to domain specific semantics has been presented in [9]. These observations also hold for other types of formalisms. For example, if we had used λ-calculus to formalize constraints, then we would not be able to decide if two sets of constraints are equivalent, as this is undecidable in λ-calculus.

2.2 Semantics and Model Transformations

Domains carry meaning beyond that of structure. For example, the model in Figure 2 describes a computational apparatus that operates on a continuous stream of data. Though the meaning of this diagram may appear obvious because of the way the model is drawn, we cannot rely on this obviousness as a definition of how a model defines a system. Instead, we must explain precisely how DSP models define computational systems. This is typically done by specifying a code generator that produces an implementation from a model. A code generator might map models from the DSP domain to models of a C++ domain. Thus, meaning is affixed to a domain by specifying a mapping from models in one domain to models in another domain. We call such a mapping an *interpretation*.

Definition 3. An *interpretation* $[\![\ ]\!]$ is a mapping from the models of one domain to the models of another domain.

$$[\![\ ]\!] : R_\Upsilon \mapsto R_{\Upsilon'} \tag{4}$$

A single domain may have many different interpretations, and these form a family of mappings $([\![\ ]\!]_i)_{i \in I}$. For some model $r \in R_\Upsilon$, we denote the i^{th} interpretation of r as $[\![r]\!]_i$. The interpretations capture the *semantics* of a domain. For example, verification tools map a non-trivial class of models onto the boolean domain $\{\texttt{true}, \texttt{false}\}$. We can think of this verification tool as an interpretation $[\![\ ]\!]_{Verify}$ that maps models onto a domain containing exactly two models. Similarly, simulators map models onto execution traces. The set of all traces can be collected together into a domain of well-formed traces, and a simulator can be expressed as the mapping $[\![\ ]\!]_{Sim}$ onto this trace domain. (Trace domains often have interesting constraints that separate the well-formed traces from the malformed ones [5].) Our approach also shows that there is no difference between semantics and model transformations. Any framework that supports model transformations also supports specification of semantics. Finally, notice that Definition 3 is weak because interpretations are defined over all model realizations, which may include malformed models, but we will show how this can be strengthened. We can now define a DSML:

Definition 4. A *domain specific modeling language* (DSML) L is a four tuple comprised of its domain and a (possibly empty) set of interpretations.

$$L = \left\langle\ \Upsilon,\ R_\Upsilon,\ C,\ ([\![\ ]\!]_i)_{i \in I}\ \right\rangle. \tag{5}$$

This definition differs from those presented elsewhere [7] because we expose the components of the structural semantics, while ignoring all together the "concrete syntax". But, other than this emphasis, there is little conceptual difference between our definition and others.

Every domain has at least one interpretation, which is its *structural interpretation*. Let Υ_B contain two nullary (arity 0) function symbols $\{true, false\}$, and let the set of well-formed models be $\{\ \{true\}, \{false\}\ \}$. The structural interpretation of a domain $[\![\ ]\!]_{struc}$ is a mapping onto R_{Υ_B} according to:

$$\begin{array}{c}([\![r]\!] = \{true\}) \Leftrightarrow (r \models C) \\ (r \not\models C) \Leftrightarrow ([\![r]\!] = \{false\}).\end{array} \tag{6}$$

The structural interpretation maps a model r to the *true* model if r satisfies its structural constraints. Otherwise r is mapped to $false$.

The framework of formal logic can also be used to specify interpretations. Recall that a model is just a set of variable-free definite clauses. Given a model r and some logic statements τ, we can deduce more definite clauses from $r \wedge \tau$. If τ is correctly defined, then the new variable-free definite clauses derived from $r \wedge \tau$ are the transformed model. We will make this more precise by first defining a *transformation*.

Definition 5. A *transformation* T is a three tuple:

$$T = \langle \Upsilon, \Upsilon', \tau \rangle \tag{7}$$

where Υ, Υ' are sets of n-ary function symbols, and τ is a set of logical statements.

A model $r \in R_\Upsilon$ is transformed to a model $r' \in R_{\Upsilon'}$ by first combining r with τ ($r \wedge \tau$), and then by generating all possible deductions from this set of sentences. The resulting set of deductions is projected onto the function symbols of Υ', producing a model purely in $R_{\Upsilon'}$. This can be described more precisely in terms of fixed-points.

Definition 6. Given a transformation T, a *transformational interpretation* $[\![\ ]\!]^T$ is a mapping:

$$\begin{array}{c}[\![\ ]\!]^T : R_\Upsilon \mapsto R_{\Upsilon'} \\ [\![r]\!]^T = (\ \psi \cap \mathcal{H}(\Upsilon', \Sigma)\ ),\end{array} \tag{8}$$

where ψ is a maximum set such that $(r \wedge \tau) \vdash \psi$.

The set ψ is a fixed-point generated by repeated application of the relevant inference procedure $\vdash$. The properties of

ψ depend on the style of logic for τ. For example, cyclic logic or logic with recursion may yield an infinite number of deductions. Non-monotonic logics may yield multiple fixed-points. To avoid these problems, transformations should be specified with a set of acyclic statements in a monotonic logic, but this is not a requirement of our abstract formalism.

Interpretations that preserve the structural semantics of domains are particularly important to embedded system design. An interpretation preserves the structural semantics if, whenever a model r is well-formed, the transformed model $[\![r]\!]^T$ is also well-formed:

$$\forall r \in R_\Upsilon, (r \models C) \Rightarrow ([\![r]\!]^T \models C') \tag{9}$$

Correct-by-construction (CbC) design is a design approach wherein structurally correct models are also behaviorally correct models with respect to certain properties. For example, the time-triggered language Giotto [21] requires a system to be described as a set of harmonically periodic tasks that communicate through single place buffers. This class of systems is provably free from communication deadlock, so designs meeting these well-formedness rules need not be checked for deadlock freedom. Of course, there are non-periodic systems without communication delays that are also deadlock free, but these systems require hard analysis algorithms and cannot be specified in a language like Giotto. Therefore, CbC tools exclude some good systems, considering them to be improper, as shown in Figure 3. Structure

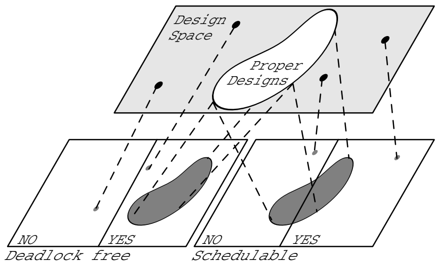

Figure 3: A correct-by-construction DSML.

preserving maps into CbC DSMLs inherit the properties of the DSML. If a structure preserving map $[\![\ ]\!]^T$ transforms models from a domain D to the Giotto domain D_{Giotto}, then every well-formed r is guaranteed to be deadlock free as interpreted by $[\![\ ]\!]^T$. This observation is significant because it means that structure preserving maps can become *property preserving maps* (where "property" implicitly means an important behavioral property of the system). For restricted classes of logic, we can automatically verify if a map is structure preserving. If structure implies behavioral correctness, as is the case for CbC DSMLs, then our verification procedures are equivalent to verifying property preservation.

2.3 Metamodels and Metamodeling

DSML structures and interpretations provide the most basic foundations for model-based embedded system design. 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*. For example, GME supports a metamodeling language, called MetaGME, based on UML class diagrams. 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.

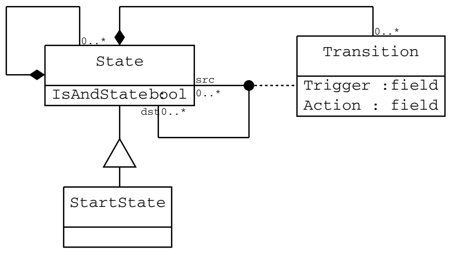

Figure 4: MetaGME metamodel for HFSM.

Figure 4 shows a MetaGME metamodel for hierarchical finite state automata. The boxes in the model are class definitions, and class members are listed under the class names. For example, the `Transition` class has `Trigger` and `Action` members, 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 `State` 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 ($\triangle$) 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 [13], 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 [19]. 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. A metamodeling language L_{meta} is a DSML with a special interpretation $[\![\ ]\!]_{meta}$ (called the *metamodeling semantics*) that maps *models* to *domains*:

$$L_{meta} = \langle \Upsilon_{meta}, R_{\Upsilon_{meta}}, C_{meta}, ([\![\ ]\!]_{struc}, [\![\ ]\!]_{meta}) \rangle \tag{10}$$

The interpretation $[\![\ ]\!]_{struc}$ is the usual structural semantics that indicates if a metamodel r is a well-formed model. If r is well-formed, then $[\![r]\!]_{meta}$ maps r to a new domain. There is one technical caveat: Interpretations map from models of one domain to models of another domain. In order to make a mapping from models to domains, we need to create a *domain of domains*. In another words, we must create a structural semantics for domains. We will show this in more detail in Section 3.

Though our formal specifications are not necessarily any less tedious to define, they serve as more than just definitions. For example, given two different metamodeling languages L_{meta} and L'_{meta} we can sometimes translate a metamodel in L_{meta} to an equivalent metamodel in L'_{meta} [17]. When this can be done automatically, domains and models can be correctly exchanged between competing tools. If $[\![\;]\!]_{meta}$ is defined transformationally, then the definition can be automatically turned into an implementation according to Equation 8. Some metamodeling languages are *metacircular*, meaning there exists a metamodel r_m such that $[\![r_m]\!]_{meta}$ is the same domain as the metamodeling language. The model r_m is called a *meta-metamodel*, and it can serve as a concise piece of documentation for the metamodeling language. Our approach also leads to algorithms for automatically proving the existence of a meta-metamodel and for deriving a meta-metamodel.

2.4 Summary

Figure 5 shows a modern view of embedded systems design that is prototypical of platform-based design [6], [16]. The middle plane (Plane II) contains a DSP model with an associated simulation semantics. This semantics is just one of the many interpretations of the DSP model. Another interpretation is obtained by transforming the DSP model into a concurrent automata model, as shown in bottom plane (Plane III). If this transformation is structure preserving, and the target class of automata have known good properties, then the DSP model also inherits these properties through the model transformation. The DSP and automata domains are concisely defined with metamodels shown in the top plane (Plane I). The arrows from the metamodels to the lower planes indicate that the domains within these planes were generated by application of the metamodeling semantics $[\![\;]\!]_{meta}$. All of the components in Figure 5 are unified into a coherent mathematical whole using our formalization of DSMLs. In the next section we will derive a version of this formalization based on a restriction of first order logic.

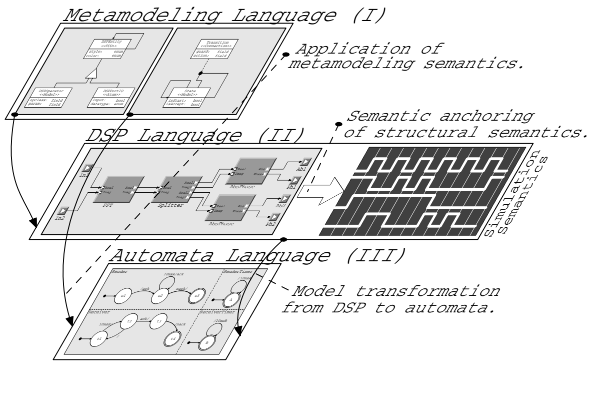
Metamodeling Language (I)
Application of metamodeling semantics.
Semantic anchoring of structural semantics.
DSP Language (II)
Simulation Semantics
Automata Language (III)
Model transformation from DSP to automata.

Figure 5: Modern embedded systems design.

3. DSMLS WITH HORN LOGIC

In this section we will develop a version of our formalism where we require well-formedness to be decidable in *polynomial time*. This is a reasonable requirement, because without this constraint any decision problem (e.g. deadlock freedom, schedulability, etc...) can be made into a well-formedness rule. Intractable well-formedness rules would lead to domains wherein large models cannot be constructed. This would break the scalability of model-based design. The formalism we develop is based on a restricted form of first order logic called Horn logic.

Horn logic is a subset of first order logic with well-known polynomial time inference procedures, so domains defined with Horn logic will be scalable. A Horn clause is a disjunction of literals with at most one non-negated literal. Horn clauses can be written in the familiar *implicative normal form* with implicit universal quantification over variables:

$$L_1 \wedge L_2 \wedge \ldots \wedge L_n \Rightarrow H \tag{11}$$

The literals $L_1, \ldots, L_n$ are called the *tail* of the clause. The literal H is called the *head* of the clause. We further restrict the logic to be acyclic, meaning that there does not exist a sequence of sentences such that[3]:

$$\begin{array}{c} L^1_1 \wedge L^1_2 \wedge \ldots \wedge L^1_{m_1} \wedge H^n \Rightarrow H^1 \\ L^i_1 \wedge L^i_2 \wedge \ldots \wedge L^i_{m_i} \wedge H^{i-1} \Rightarrow H^i \\ L^n_1 \wedge L^n_2 \wedge \ldots \wedge L^n_{m_n} \wedge H^{n-1} \Rightarrow H^n \end{array} \tag{12}$$

Cycles of this form a similar to algebraic loops; in logic they can lead to an infinite number of conclusions, so they are not suited for our purposes. Horn logic is already implemented in programming languages like Prolog, usually with the SLD resolution algorithm [3]. For simple problems, like checking well-formedness, we can directly use these existing tools. However, most of the analysis problems we encounter require more sophisticated tools. For these tasks we have developed new tools specifically for analyzing DSMLs. We will now concretize our formalism with Horn logic.

3.1 Domains and Structural Semantics

Domains defined with Horn logic inherit all of the formalism that we previously described. In this section we show how to define the DSP domain with Horn logic, which means writing the constraints in the proper form. There are a number of constraints on the DSP domain. For example, legal DSP models do not have dataflow connections that start (end) on system outputs (inputs). (This is not true for interfaces.) The following rules capture this malformedness criteria:

$$\begin{array}{c} flow(x_1, y_1, x_2, insig(y_2)) \Rightarrow malform(insig(y_2)) \\ flow(x_1, outsig(y_1), x_2, y_2) \Rightarrow malform(outsig(y_1)) \end{array} \tag{13}$$

Unfortunately, not all constraints can be written as pure Horn clauses. For example, each instance of a DSP primitive ($inst(x, y)$) must identify a properly defined DSP primitive ($y = prim(y')$). If we define the positive constraint $inst(x, prim(y)) \wedge prim(y) \Rightarrow wellform(x)$, then models with at least one correctly declared instance will be identified as correct. The constraints should express that *every* occurrence of $inst(x, prim(y))$ must have an occurrence

[3]The superscript is the sentence number and the subscripts order the terms within a sentence

of $prim(y)$, but this requires a unique constraint for each $inst(x, prim(y))$ and makes the set C dependent on r.

We must extend Horn logic in order to handle these types of constraints, and two well-known extensions are possible: lists and negation-as-failure. Lists allow smaller proofs to be accumulated into larger proofs. For example, if we could build a list of all occurrences of $wellform(x)$ and a list of all occurrences of $inst(x, prim(y))$, then the model would be well-formed if the lists were the same length. This extension works well for proving well-formedness of a given model r. However, our goal is also to find the existence of models that satisfy certain properties, and lists significantly complicate existence proofs because their lengths cannot be bounded.

The other option is to introduce a form of negation so that the constraint would become $inst(x, prim(y)) \wedge \neg prim(y) \Rightarrow malform(x)$. In another words, a model is malformed if for some $inst(x, prim(y))$ there does not exist a corresponding $prim(y)$. The negation used here is not typical negation, but *negation-as-failure* (NAF). In order to distinguish NAF from true negation, we will use the (non-standard) symbol !. Negation-as-failure means that $!L$ is true if the positive literal L cannot be proved. Negation-as-failure has a significant disadvantage: If used improperly, the proof techniques of Horn logic become unsound and can make incorrect deductions. In order to avoid this, we permit a restricted form of NAF, though we will not prove its soundness here:

Remark 1. Given r, C, (with $r \wedge C$ acyclic) and a sentence $(\ldots, !L_i, \ldots, L_j, \ldots \Rightarrow H) \in C$, the following properties must hold[4]:

1. There exists some variable x in $!L_i$ such that x is also in a positive literal L_j, $(i \neq j)$
2. If variable y is in $!L_i$ and y is not in a positive literal L_j, then y cannot not appear in another negative literal $!L_k$, $(k \neq i)$.

Though NAF must be used carefully, it does not lead to unbounded search spaces, so analysis problems can be solved. The following are some constraints for the DSP language using negation-as-failure.

1. Instances must use primitives that are defined: $inst(x, prim(y)) \wedge !prim(y) \Rightarrow malform(x)$,
2. Ports are placed on defined primitives: $iport(prim(x), y) \wedge !prim(x) \Rightarrow malform(y)$,
3. Dataflow connections must start on defined ports: $flow(x, oport(prim(y), z), u, w) \wedge !oport(prim(y), z) \Rightarrow malform(z)$.

We will encounter important analysis problems for Horn domains. Most commonly, we must decide if two Horn domains define the same set of models. Let Υ_C be all the function symbols used to define the constraints C. Given two domains with (Υ, C) and (Υ', C'), we rewrite the constraints C, C' to $\overline{C}, \overline{C}'$, such that:

$$((\Upsilon_{\overline{C}} \cup \Upsilon) \cap (\Upsilon_{\overline{C}'} \cup \Upsilon')) - (\Upsilon \cap \Upsilon') = \emptyset \tag{14}$$

In another words, we rewrite the constraints so that the only symbols in common are those of the model primitives. Assuming that both domains are Negative Horn Domains, we proceed to find a model r that is in the first domain and not in the second: $(\overline{C} \wedge \overline{C}' \wedge r) \vdash (!malform_1(x) \wedge malform_2(y))$. If no such r exists then we know that $D(\Upsilon, C) \subseteq D(\Upsilon', C')$. We then try to find a model r' that is in the second domain and not in the first: $(\overline{C} \wedge \overline{C}' \wedge r') \vdash (!malform_2(x) \wedge malform_1(y))$. If neither r nor r' exists, then the domains must define the same set of models. This method is similar to methods for proving the equivalence of two formal languages. We use $D \cong D'$ to denote equivalence of two domains, and $C \cong C'$ to denote equivalence of two constraint sets.

3.2 Semantics and Interpretations

We use Horn logic to define transformational interpretations by writing transformations as a set of Horn sentences. The set ψ of Equation 8 becomes the fixed-point of the *forward chaining procedure*, which is a well-known algorithm for generating all of the conclusions from a set of Horn sentences. The fixed-point ψ is unique because the logic is monotonic[5] and it can be computed in finite time because the logic is acyclic.

We now illustrate Horn interpretations with one of the simplest composition semantics, the asynchronous (shuffle) product of finite state automata. Given two automata $\mathcal{A}_1, \mathcal{A}_2$, each with a set of states Q_i, a set of initial states Q_i^0, a finite alphabet of events Σ_i, and a transition relation $\longrightarrow$, the asynchronous product forms a product automata such that no transition from $\mathcal{A}_1$ is simultaneous with a transition from $\mathcal{A}_2$. The rules for composition are:

$$\begin{gathered}\Sigma_{1,2} = \Sigma_1 \cup \Sigma_2,\ Q_{1,2} = Q_1 \times Q_2,\ Q_{1,2}^0 = Q_1^0 \times Q_2^0 \\ (s_1 \xrightarrow{\alpha}_1 s_1') \wedge s_2 \Rightarrow (s_1, s_2) \xrightarrow{\alpha}_{1,2} (s_1', s_2) \\ s_1 \wedge (s_2 \xrightarrow{\alpha}_2 s_2') \Rightarrow (s_1, s_2) \xrightarrow{\alpha}_{1,2} (s_1, s_2')\end{gathered} \tag{15}$$

Before we can give the Horn interpretation, we must give an encoding of automata so that the interpretation has a well-defined domain/codomain. Let $s_i(x)$ denote that x is a state of $\mathcal{A}_i$, $initial_i(x)$ denote that x is an initial state of $\mathcal{A}_i$, $event_i(x)$ denote that x is an event of $\mathcal{A}_i$, and let $e_i(x, a, x')$ denote $x \xrightarrow{a}_i x'$. Let the input domain be defined by:

$$\Upsilon = \left\{ \begin{array}{c} s_1(x), s_2(x), initial_1(x), initial_2(x), \\ event_1(x), event_2(x), e_1(x, a, x'), e_2(x, a, x') \end{array} \right\} \tag{16}$$

and the output domain be defined by:

$$\Upsilon' = \left\{ \begin{array}{c} s_{1,2}(x), initial_{1,2}(x), \\ event_{1,2}(x), e_{1,2}(x, a, x') \end{array} \right\} \tag{17}$$

The asynchronous (shuffle) product of automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is given by the Horn transformation $T_{async} = \langle \Upsilon, \Upsilon', \tau \rangle$ where:

$$\tau = \left\{ \begin{array}{c} event_1(x) \Rightarrow event_{1,2}(x), \\ event_2(x) \Rightarrow event_{1,2}(x) \\ s_1(x) \wedge s_2(y) \Rightarrow s_{1,2}(x, y) \\ initial_1(x) \wedge initial_2(y) \Rightarrow initial_{1,2}(x, y) \\ e_1(x, a, x') \wedge s_2(y) \Rightarrow e_{1,2}((x, y), a, (x', y)) \\ s_1(x) \wedge e_2(y, a, y') \Rightarrow e_{1,2}((x, y), a, (x, y')) \end{array} \right. \tag{18}$$

We now apply this transformation to the two example au-

[4]There may be additional requirements on ordering imposed by the particular NAF implementation.

[5]This claim is not always true when using NAF, but we will not discuss this issue here.

tomatons in Figure 6. Automaton $\mathcal{A}_1$ is encoded as:

$$r_{\mathcal{A}_1} = \left\{ \begin{array}{l} event_1(\mathtt{e}_1),\ event_1(\mathtt{e}_2), \\ initial_1(\mathtt{A}),\ s_1(\mathtt{A}),\ s_1(\mathtt{B}),\ s_1(\mathtt{C}), \\ e_1(\mathtt{A},\mathtt{e}_1,\mathtt{B}),\ e_1(\mathtt{A},\mathtt{e}_2,\mathtt{C}) \end{array} \right. \tag{19}$$

Automaton $\mathcal{A}_2$ is encoded as:

$$r_{\mathcal{A}_2} = \left\{ \begin{array}{l} event_2(\mathtt{e}_3),\ initial_2(\mathtt{D}), \\ s_2(\mathtt{D}),\ s_1(\mathtt{E}), e_2(\mathtt{D},\mathtt{e}_3,\mathtt{E}) \end{array} \right. \tag{20}$$

The forward chaining of $(r_{\mathcal{A}_1} \wedge r_{\mathcal{A}_2}) \wedge \tau$ yields the following additional deductions:

$$r_{\mathcal{A}_{1,2}} =$$

$$\left\{ \begin{array}{l} event_{1,2}(\mathtt{e}_1),\ event_{1,2}(\mathtt{e}_2),\ event_{1,2}(\mathtt{e}_3), \\ initial_{1,2}(\mathtt{A},\mathtt{D}),\ s_{1,2}(\mathtt{A},\mathtt{D}),\ s_{1,2}(\mathtt{A},\mathtt{E}), \\ s_{1,2}(\mathtt{B},\mathtt{D}),\ s_{1,2}(\mathtt{B},\mathtt{E}),\ s_{1,2}(\mathtt{C},\mathtt{D}), \\ s_{1,2}(\mathtt{C},\mathtt{E}),\ e_{1,2}((\mathtt{A},\mathtt{D}),\ \mathtt{e}_1,(\mathtt{B},\mathtt{D})), \\ e_{1,2}((\mathtt{A},\mathtt{D}),\mathtt{e}_2,(\mathtt{C},\mathtt{D})),\ e_{1,2}((\mathtt{A},\mathtt{D}),\mathtt{e}_3,(\mathtt{A},\mathtt{E})), \\ e_{1,2}((\mathtt{B},\mathtt{D}),\mathtt{e}_3,(\mathtt{B},\mathtt{E})),\ e_{1,2}((\mathtt{C},\mathtt{D}),\mathtt{e}_3,(\mathtt{C},\mathtt{E})), \\ e_{1,2}((\mathtt{A},\mathtt{E}),\mathtt{e}_1,(\mathtt{B},\mathtt{E})),\ e_{1,2}((\mathtt{A},\mathtt{E}),\mathtt{e}_2,(\mathtt{C},\mathtt{E})) \end{array} \right. \tag{21}$$

The deductions in $r_{\mathcal{A}_{1,2}}$ are all in the Herbrand Universe of the target domain, $\mathcal{H}(\Upsilon', \Sigma)$, so $r_{\mathcal{A}_{1,2}}$ is the asynchronous interpretation $[\![r_{\mathcal{A}_1} \wedge r_{\mathcal{A}_2}]\!]^{T_{async}}$. The reader can more easily verify this by considering the graphical rendering of $r_{\mathcal{A}_{1,2}}$ in Figure 7.

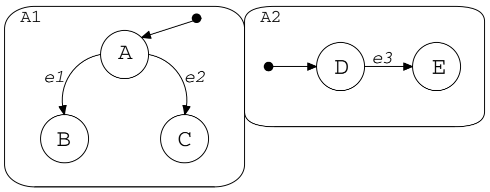

Figure 6: Two example automata.

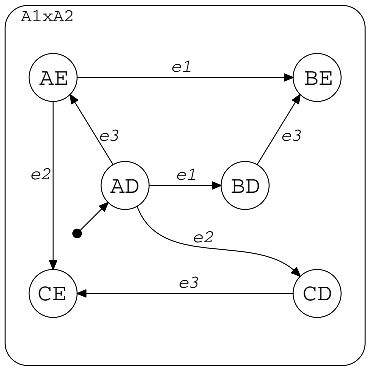

Figure 7: Asynchronous product via $[\![\ ]\!]^{T_{async}}$.

3.3 Metamodeling Semantics

Correctly formalizing metamodeling requires us to address a technical, but important issue. Interpretations map models to models, but the metamodeling semantics maps models to domains, which are (infinite) sets of models. On the surface it would appear that our notion of an interpretation cannot handle this concept. However, we also know that our domains are defined by a finite set of symbols along with a finite set of constraints. The key is to exploit the fact that domains can be finitely represented, and then to build a domain where finite models are isomorphically related to finite sets of constraints. We will call this domain D_{Horn}, because it must represent domains with constraints written as Horn clauses. Models that are transformed onto D_{Horn} can be related to a domain through this isomorphism. Semantics built from other logical styles require developing other isomorphisms, but we do not believe this to be difficult.

Before we define D_{Horn}, we will build the reader's intuition by showing the conversion of a set of Horn Clauses into a model that only has variable-free definite clauses. We use this following isomorphism δ to convert clauses into models.

Definition 7. Let Υ^* be a set of n-ary function symbols such that $\{constraint, neg, var\} \subset \Upsilon^*$. The *domain representation function* δ is a mapping from a set of Horn sentences C to a set of variable-free definite clauses according to the following structural induction:

1. Let $s_i \in C$, then $\delta(s_1, s_2, \ldots, s_m) = \bigwedge_i \delta(s_i)$
2. $\delta(\ L_1, \ldots, !L_i, \ldots, L_n \Rightarrow H\ ) =$ $constraint(\delta(H), \delta(L_1), \ldots, neg(\delta(L_i)), \ldots, \delta(L_n))$
3. $\delta(\ f(\ldots, x, \ldots)\ ) = f(\delta(\ldots), var(\mathtt{x}), \delta(\ldots))$, where x is a variable, $\mathtt{x}$ is a constant in Σ^*, and $f \in \Upsilon^*$.
4. $\delta(\ f(\ldots, \mathtt{c}, \ldots)\ ) = f(\delta(\ldots), \mathtt{c}, \delta(\ldots))$, where c is a constant in Σ^*.

This definition assumes that each sentence is the same length, but this is not a problem because sentences can be padded with the nullary function *true*. Consider some of the constraints from the DSP language:

$$C = \begin{array}{c} flow(x_1, y_1, x_2, insig(y_2)) \Rightarrow malform(insig(y_2)), \\ inst(x, prim(y)) \wedge !prim(y) \Rightarrow malform(x) \end{array}$$

According to the structural induction of Definition 7, we conclude that:

$$\delta(C) =$$
$$\begin{array}{l} constraint\left(\begin{array}{l} malform(insig(var(\mathtt{y}_2))), flow(var(\mathtt{x}_1), \\ var(\mathtt{y}_1), var(\mathtt{x}_2), insig(var(\mathtt{y}_2))), true \end{array} \right), \\ constraint\left(\begin{array}{l} malform(var(\mathtt{x})), inst(var(\mathtt{x}), \\ prim(var(\mathtt{y}))neg(prim(var(\mathtt{y}))) \end{array} \right) \end{array}$$

If we make a simplifying assumption that the symbols *neg*, *var*, and *constraints* are only used to encode Horn sentences, and are not used in any other domains, then it is easy to see that the inverse δ^{-1} exists and maps models back to sets of constraints. (We do not need to make this assumption.) From the encoding given by δ, we also see that the set of function symbols Υ^* must contain the function symbols of all domains. We therefore construct Υ^* so that it contains all possible function symbol names Σ^*[6]. The well-formedness rules of the domain eliminate models that do not encode to well-formed Horn sentences. We will not describe all of these rules here.

We can now define a metamodeling semantics transformationally by writing a transformation T_{meta} from metamodels to models of the Horn domain D_{Horn}. The actual domain is

[6]This does not correctly handle the arity of functions. Handling this requires a more complicated encoding that detracts from the simplicity of the δ shown here.

recovered by applying δ^{-1}. We abbreviate this process with the notation *meta*(r), where r is a well-formed metamodel.

$$meta(r) \mapsto \langle \Upsilon_C,\ R_{\Upsilon_C},\ \delta^{-1}(\llbracket r \rrbracket^{T_{meta}}) \rangle \tag{22}$$

Several important observations come from our formalization of the metamodeling process. Two metamodels are equivalent if the domains they define are equivalent:

Definition 8. Given two metamodels r and r', the metamodels are said to be *equivalent*, written $r \cong r'$ if:

$$\delta^{-1}(\llbracket r \rrbracket^{T_{meta}}) \cong \delta^{-1}(\llbracket r' \rrbracket^{T_{meta}}) \tag{23}$$

The process for checking this equivalence was given in Section 3.1. Another important property is *metacircularity.* UML [20], MOF, and MetaGME all assume this property, but it has never been formally verified in the literature. With Horn logic, we can formally define and check for this property:

Definition 9. A metamodeling language L_{meta} is *metacircular* if:

$$\exists r_m \in D_{meta},\ \delta^{-1}(\llbracket r_m \rrbracket^{T_{meta}}) \cong C_{meta} \tag{24}$$

The model r_m is called the *meta-metamodel.*

Proving metacircularity is harder than proving equivalence of two metamodels, because the search algorithms that find r_m must account for all possible encodings of C_{meta} in D_{Horn}. In another words, it is not sufficient to search for metamodels that map to $\delta(C_{meta})$; one must also search for metamodels that map to any other $\delta(C')$ such that $C' \cong C_{meta}$. This problem is decidable if additional care is taken in defining the encoding δ and in writing the transformation T_{meta}. However, we will not discuss these details further.

Our formalism also has interesting implications on current standards like MDA, UML, MOF, and MetaGME. To varying degrees, these standards use the term meta-metamodel synonymously with "the definition of the metamodeling language". However, we have shown that a meta-metamodel is not a definition of the metamodeling semantics. Rather, it is a consequence of the metamodeling semantics, and this is why it can be automatically discovered. This recognition is more profound than just misuse of terminology, because today's metaprogrammable modeling tools are hard-coded with a particular metamodeling language. If the metamodeling semantics is viewed as just another model transformation, then there becomes no reason to hard-code a tool around a particular meta-metamodel. The fundamental concepts that should be fixed are the way primitives are composed into models (i.e. via the Herbrand Universe) and the style of logic used to write constraints and transformations. Tools built up from this foundation could simultaneously support many different metamodeling languages, and new metamodeling languages could be arbitrarily created without rewriting the tool. Even without rebuilding tool infrastructure, metamodels should be viewed as formal entities, and as such, it should be possible migrate them across different tools while preserving their structural semantics.

3.4 Analysis and Verification

Unfortunately we have little space to discuss all the details of analysis and verification. We will try to briefly outline the approach we have developed in our tool *FORMULA* (Formal Modeling Using Logic Analysis). First, let us describe in more detail the problems that can be solved in languages like Prolog. Logic programming languages make the fundamental assumption, called the *Closed World Assumption* (CWA), that all information known about the world is explicitly stated. CWA means that we can prove anything about a given model. In fact, with Horn logic, we can prove anything in polynomial time. Thus, classical tools work perfectly well for answering queries about a single model, such as "Is the model well-formed?".

More complicated analysis problems, like checking equivalence of domains, require finding the existence of a world that satisfies a certain property. Tools that handle these queries must know about the (infinite) set of all worlds and the constraints that bound this set. Clearly, this set must be searched carefully, otherwise algorithms can easily become undecidable. Classical logic programming is not suited to solve these problems, though these languages can be used as a foundation to implement the necessary algorithms. However, logic programming languages must be used carefully because they usually come with a rich set of built in functions. For example in Prolog one may write:

```
f(Y) :- a(X), Y is 4*X;
g(Y) :- b(X), Y is 2*X;
```

If we wish to find a model r such that $r \vdash f(Y) \wedge g(Y)$, then we must solve the linear equation $4x_1 = 2x_2$. An analysis algorithm that handles these innocent looking functions would require a complete numerical solver. FORMULA does not handle these built in functions but it does correctly handle NAF and disequalities ($\neq$).

FORMULA is given a list of function symbols, function arities, Horn sentences, and a property ϕ. It then finds a model r containing only the function symbols of Υ such that $r \vdash \phi$. The function symbols are divided into three types *in*, *closed*, and *private.* Function symbols marked *in* are members of Υ. Function symbols marked *closed* are members of Υ, but their possible values are already given. If the user marks all symbols of Υ *closed* , then the user forces CWA. Function symbols marked *private* are used only for calculating constraints and transformations. They will not be included in the final solution.

Working backwards from the property ϕ, FORMULA constructs a backwards chaining tree that terminates at function symbols in Υ. If an Υ symbol is reached, and it contains variables that are not constrained to a constant, then each variable is given a unique value from Σ^* that does not appear anywhere else. If a NAF literal $!L_i$ is encountered, then all permutations of the tail of L_i and any heads that unify with L_i that do not prove L_i are added to the backwards chaining tree. If not proving L_i requires not proving some other literal L', then a constraint $!L'$ is added to the tree. Whenever this constraint is enabled, it applies globally to all past and future deductions, and it may cause failure of a possible solution. In essence, this converts the NAF to a form of *constructive negation* that does not suffer from the soundness problems of NAF [4]. For this reason, we can also solve for properties that violate the previous restrictions we placed on negation (e.g. $r \vdash malform_1(X) \wedge !malform_2(Y)$). FORMULA also implements backtracking and will search from every possible restart location before failing. FORMULA contains built in functions for renaming sets of constraints, for generating the possible encodings of a set of Horn sentences in the domain D_{Horn}, and for converting back and forth from D_{Horn} and sets of Horn sentences. In summary, it provides a powerful formal foundation for analyzing Horn-based DSMLs.

4. DISCUSSION AND CONCLUSION

The aim of this paper was to present our formal foundation for DSMLs. Unfortunately, this has not left us room to show how this formalism is used with existing tools. In parallel with our formalism, we have been developing a tool suite called 4*ml* (pronounced *formal*) that interfaces with GME, MetaGME, and the graph rewriting tool GReAT. Currently we can import and export existing GME models and metamodels with the tool *GME-*4*mlizer*. After a GME model is converted to a set of definite clauses it can undergo well-formedness checking and transformations. We are currently working on formalizing all of MetaGME's metamodeling semantics so that every existing metamodel can be transformed into its corresponding formal definition. Additionally, we are developing a tool, called the *GReAT-*4*mlizer*, that converts a subset of GReAT graph transformations into Horn transformations. This process has the added benefit of formalizing *attribute mappings*, which are currently informal in GReAT. As previously described, FORMULA is the centerpiece tool for analysis of DSMLs. This formalization also opens up avenues for new tools. For example, given a polynomial time structural semantics, we can generate domain specific real-time constraint checking engines. Such engines are useful for embedded systems that take models as input or modify internal models in real-time. These engines can be deployed independently of GME and optimized for the particular domain from which they were generated. Our ongoing work is to produce a formal backplane on top of which existing tools will sit. More detailed information on our tool suite can be found at [1].

This work has evolved naturally from our previous work on using separation of concerns for embedded system design [14] and on the semantic anchoring of DSMLs [15]. Both of these research strands identified the need for a rigorous understanding of the modeling and metamodeling process. In [14], we identified structural constraints that conservatively approximate behavioral properties of synchronous reactive systems in polynomial time. We now understand that we actually defined two Horn domains, one with correct-by-construction properties, and a structure preserving map between these domains. In the semantic anchoring work of [15] we transformed domains into ASM *datamodels*, but we lacked a precise definition of domains and transformations. With this formalism, we now understand these concepts and we hope to leverage them to create a more comprehensive union between structural and behavioral semantics.

5. REFERENCES

[1] http://www.isis.vanderbilt.edu/projects/4ml.

[2] A. Ledeczi, M. Maroti, A. B. G. K. J. G. C. T. I.-G. N. J. S. P. V. The generic modeling environment. *Workshop on Intelligent Signal Processing* (May 2001).

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

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

[5] E. Lee, A. S.-V. A unified framework for comparing models of computation. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems 17*, 12 (December 1998), 1217–1229.

[6] F. Balarin, Y. Watanabe, H. H. L. L. C. P. A. L. S.-V. Metropolis: an integrated electronic system design environment. *IEEE Computer 36*, 4 (April 2003).

[7] G. Karsai, J. Sztipanovits, A. L. T. B. Model-integrated development of embedded software. *Proceedings of the IEEE 91*, 1 (January 2003), 145–164.

[8] G. Karsai, A. 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.

[9] Gupta, G. Horn logic denotations and their applications. *The Logic Programming Paradigm: A 25 year perspective*, 127–160.

[10] Gurevich, Y. Evolving algebra: An attempt to discover semantics. *EATCS 43* (1991), 264–284.

[11] Herbrand, J. *Logical Writings*. Harvard University Press, Cambridge, MA, 1971. Edited by Warren D. Goldfarb.

[12] Horn, A. On sentences which are true on direct unions of algebras. *Journal of Symbolic Logic 16* (1951), 14–21.

[13] Institute For Software Integrated Systems. Gme 5 user's guide. Tech. rep., Vanderbilt University, 2005.

[14] Jackson, E. K., and Sztipanovits, J. Using separation of concerns for embedded systems design. *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)* (September 2005), 25–34.

[15] K. Chen, J. Sztipanovits, S. N. M. E., 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).

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

[17] M. Emerson, J. Sztipanovits, T. B. A mof-based metamodeling environment. *Journal of Universal Computer Science 10*, 10 (October 2004), 1357–1382.

[18] Mal'cev, A. I. The metamathematics of algebraic systems. *Studies in Logic and The Foundations of Mathematics 66* (1971).

[19] Object Management Group. Meta object facility specification v1.4. Tech. rep., 2002.

[20] Object Management Group. Unified modeling language: Superstructure version 2.0, 3rd revised submission to omg rfp. Tech. rep., 2003.

[21] T. A. Henzinger, C. M. Kirsch, M. A. S., and Pree, W. From control models to real-time code using giotto. *Control Systems Magazine 2*, 1 (2003), 50–64.