Fundamental Limitations in Domain-Specific Modeling Language Evolution

Jonathan Sprinkle, Jeff Gray, and Marjan Mernik

Abstract—In this paper we address language engineering issues surrounding domain-specific modeling languages (DSMLs). By definition, such languages track the domain, meaning that changes to the domain require changes to the DSML in order to provide an intuitive specification of domain-specific models. For this work, our primary focus is on fundamental limitations that affect the preservation of semantics during domain model evolution. We specifically address fundamental limitations in semantics-preserving transformations, and/or the implementation of algorithms that specify such transformations. This work has implications for language engineers who are planning for the maintenance of models, or designing model transformations for the purpose of preserving semantics. We provide a brief representative example from the discipline of hybrid systems, where such results can be interpreted.

Index Terms—Domain-specific modeling, model-driven development, language engineering, model transformations.

1 INTRODUCTION

The software engineering community has been significantly influenced by the Unified Modeling Language (UML) and its accompanying Unified Software Development Process [1]. Many engineering applications can benefit further from the structure, intuitive appearance, and other formalisms that domain-specific languages (DSLs) give to software developers, by enabling the formal specification of the design of the system, rather than the design of the system software.

This paper involves a subclass of design environments that integrate domain-specific modeling languages (DSMLs), where the design, implementation, constraints, semantics, patterns, and/or operation are inspired by one or more domains. DSLs and DSMLs are found in both textual [2] and graphical [3] form, and their development is subject to the same concerns as any software language, including: the definition and representation of grammar, syntax, and schema; attachment of semantics and interpretation of the language; specification and enforcement of constraints (e.g., those of OCL); and lifecycle management, including revision history and debugging. In addition to their definition, the implementation practice of DSMLs can also be stylized by similar techniques to those used in general-purpose languages (e.g., Java), including: refactoring; modularization; code synthesis (especially generative programming); and design patterns.

The ability to rapidly define DSMLs [4] enabled domain experts to use rapid prototyping in a language close to the domain. Domain model evolution is common in DSMLs, since the ease of changing the DSML based on customer feedback—as well as evolution in the domain itself—enables such changes to occur incrementally and often. When existing models no longer conform to the evolved language, developers must either (1) rebuild or (2) modify the existing models to conform to the new language. Either solution must also address the semantics of models in the new language, compared to the original models.

Domain evolution, especially domain model evolution driven by changes to the DSML, is the focus of this paper. When a DSML is based on a domain, then changes in that domain motivate changes in the DSML: this will likely require changes to the models built using that language. A formal definition of domain evolution is given in Section 4; informally, domain evolution is the process through which a DSML maintains coherency with the domain it abstracts, through updates in the syntax, constraints, and semantics of that domain. We choose this topic because it provides a rich set of cases to consider the semantic equivalence of models across evolutions of the domain in the appropriate revisions of the DSML (i.e., domain
model evolution). These cases provide a microcosm for understanding semantic equivalence between models expressed in two languages. In Figure 1, an abstraction of this problem, and its motivations, is shown.

![Diagram showing changes to the domain often motivate changes to other abstractions of that domain. The formalisms used here are explained in Sections 3 and 4.](image)

In fact, our proposition is that there exist fundamental limitations that restrict the semantic translation of models expressed in a DSML to a subsequent revision based solely on the interpretation of variances in syntax. Limitations in the expression of semantics make it difficult (if not impossible) to verify that a particular model transformation will produce semantically correct translated models for any arbitrary model conforming to a DSML, even if the DSML and its evolved version are known a priori. Additionally, the introduction of constraint-based correctness makes it more problematic to prove that certain transformations represent a total solution for domain model evolution.

1.1 Scope

The contribution of this paper is that the preservation of semantics during the evolution of DSMs can be considered in several cases. Some of these cases permit automated transformation of domain models, while in others it is certain that a domain expert must be involved in order to evaluate correctness. We focus the discussions in this paper on semantics-preserving transformations due to domain evolution, specifically the fundamental limitations that are associated with automating such transformations. Thus, we do not generally consider the semantics of transformations between two disparate modeling languages. We also do not consider the techniques used to perform these transformations as relevant to the preservation of semantics, but rather depend on existing state-of-the-art research (discussed in Section 2). Importantly, we make no judgements on the UML2-style platform-independent model (PIM) to platform-specific model (PSM) transformations, because the semantics of such transformations are an onus on the transformation designer. However, we may be able to provide insight into future research directions in the automatic generation of PIM–PSM transformations, which are subject to the fundamental issues we discuss.

1.2 A note on approach

Our technical approach in this paper, of considering languages and models built in those languages within the formalisms of set theory, may seem overly formal for the results we present. We point out we use the set formalism precisely for its ability to tersely describe the kinds of evolution we consider, and the consequences can be understood by even introductory students of set theory. We compare this treatment to one of visual diagrams (which would reduce the paper’s impact in this space) or to that of prose (which requires careful readers to adopt an exact meaning to words that are frequently overloaded in the software community).

2 Background

The history of programming languages provides a rich set of examples for creating new software languages to optimize software development. We argue that regardless of the motivation of a new software language, its implementation can often be considered domain-specific based on the context of usage. That is to say, a language designer may choose keywords or atomic types, and semantics, based on the intuitive interpretation of a particular domain. As an example, the UML is a general-purpose modeling language that provides a fixed notation that can be used for many domains, but it is frequently used to specify software concerns.

The wide availability of the graphical user interface\(^1\) at the operating system level in the early

---

1. That is, a GUI as opposed to the textual user interface, though technically even a textual interface is graphical.
1990’s enabled the rise of visual languages [5]. At first, visual languages were inspired by formal design languages, such as Harel’s Statecharts [6], that were intuitive, well-defined, terse, and executable. Such formalisms paved the way for well-defined graphical languages that were a concrete representation of these formalisms. For models of computation such as dataflow, a graphical language enabled an executable version of diagrams used in the engineering process [7].

Further, an appropriate choice of concrete syntax could yield an intuitive interface for specifying the behavior of a system [8]. The formal expression of a system’s abstraction in such a visual language allowed for the birth of domain-specific modeling (DSM), which provides a user interface that is representative of the domain that is under design [9]. In the remainder of this section, we describe previous efforts and related work that lay the foundations for the results we present.

2.1 Foundational Work

The increased interest in DSM can be traced to several initial metatools that emerged during the 1980s. The System Encyclopedia Manager (SEM) is one of the earliest metatools that enabled the modeling of various categories of systems [10]. The MetaPlex tool is an example of a first-generation metaCASE tool [11] that used a textual language to define metamodels that specified new modeling environments. Another early example, which was eventually commercialized, was the ToolBuilder metaCASE system [12]. ToolBuilder consisted of a language for describing the domain-specific tool definition, with capabilities for generating domain models that executed on a specific run-time infrastructure. The ARIES environment [13] used modeling for the purposes of requirements specification, and tackled many issues also relevant to domain-specific modeling.

Although many of these seminal efforts are no longer used in practice, there are two metamodeling tools that have been available since the early 1990s and adopted in multiple industrial contexts: MetaEdit+ [9] and the Generic Modeling Environment (GME) [14]. MetaEdit+ is a commercially available metamodeling tool suite for specifying modeling concepts for a specific domain (e.g., domain properties, rules, and visual symbols). The metamodel is persistent in the MetaEdit+ repository, which allows evolutionary changes that reflect automatically to models and generators. The GME also provides a metamodeling interface that allows a language designer to describe the essential characteristics of a language using UML and OCL as the metamodeling language. In GME, both the metamodeling interface and the subsequent generated modeling language are hosted within the GME; i.e., the GME has a meta-metamodel available for defining metamodels.

A long-standing challenge in DSM is the lack of a semantic definition within the context of the metamodel. Most metamodeling environments capture abstract syntax and constraints, but the issue of domain semantics is usually embedded in one or more code generators. This provides a challenge for language evolution as there is no standard formalism for specifying the semantics. A potential solution to this challenge can be found in semantic anchoring [15], which maps the semantics of a new modeling language to the existing semantics of a well-known language.

For more information about DSM and its foundations, there is a long-running workshop series that has tracked advancements in DSM research and practice [16]. For a general overview, Kelly and Tolvanen introduce the history of DSM and present a specific tooling approach using MetaEdit+ [9]. A brief summary of language design issues specific to DSM is in [17].

2.2 Related Work

2.2.1 Heterogeneous and Multi-Paradigm Modeling

It is often desirable that DSLs (including DSMLs) represent concepts from several domains and various models of computation. For example, hybrid system modeling necessitates language formalisms that combine concepts to represent discrete behavior (e.g., state machines) with continuous behavior of system flow (e.g., differential equations) [18]. The seminal work on such heterogeneous modeling emerged from the Ptolemy project [19], which assisted in the mixing of different models of computation. Multi-paradigm modeling combines formalisms according to their unique strengths to obtain synergistic benefits useful for some domains [20]. Despite the benefits that hybrid and multi-paradigm modeling offer, the combination of different formalisms presents another challenge for DSL evolution as adaptations to each language may have coupling ramifications across the concepts that may be represented (i.e., evolving a hybrid language may require the adaptation of many separate models of computation within the language).

A growing area of research is focusing on the...
integration of semantic relationships among DSLs [21].

2.2.2 Model Transformations
Model transformation represents the conversion from one model representation (the source) into the representation of another model (the target). Much of the research and practice of model transformation has focused on exogenous translations [22] where the source and target are from different metamodels. The focus of this paper is more closely related on endogenous refinements where the source and target share the same metamodel, or a metamodel with only evolutionary changes. As a shorthand, consider that exogenous translations are concerned with transformations of the form $D_1 \rightarrow D_2$, but the endogenous refinements used in this paper are of the form $D \rightarrow D'$ transformations, meaning that $D$ and $D'$ are closely related. Surveys of model transformation research are available in [22] and [23]. The International Conference on Model Transformation (ICMT) serves as a forum that defines research advances in model transformation.

The power of model transformation within the context of MDE is key to the work of Fabro [24] and Porres [25] (the latter concentrates on model refactoring [26]). Additional insight in model refactoring can be found in Pretschner and Prenninger [27], who discuss transformations for semantically equivalent state machines. Agrawal and others produced the GReAT toolbox [28], and well-defined semantics for using graph rewriting in the specification of the semantics of a modeling language. The scientific approach of graph rewriting is a strong discipline in and of itself, as noted by Schürr in the power of triple-graph grammars [29]. We view graph rewriting as a method for developing model transformations, and its advancement improves the performance and utility of such transformations [30]. Among the several dozen examples of model transformation languages are VIATRA [31], ATL [32], and AToM³ [33].

2.2.3 Tool Integration
The integration of various tools is a common requirement throughout the software development process. It is often desirable to consult several tools that perform similar functionality in the same domain to obtain different perspectives and results to aid design and maintenance decisions [34]. In general, the challenge of integrating the results of a cadre of tools is that the output of each tool may have a proprietary format and representation. The lack of a uniform format for different tools is a common barrier toward tool interoperability [35]. Homogenizing various output formats of the tools in a given domain greatly simplifies the utilization of different tools.

There has been a surge of interest in applying model engineering and DSMLs to tool integration, with the benefits of model transformations discussed in [36]. The two issues of syntactic and semantic interoperability of tools is discussed in [37], which also advocates model transformation as the conversion mechanism between tool models. An advantage of a model transformation approach is that new tools can be added easily to the integration chain by specifying mappings to a pivot model that captures the integrated concerns of tools in a specific domain. Within the context of this paper, tool integration solutions may be subject to DSML evolution issues. A detailed list of the various types of changes that may occur from domain evolution or user requests is listed in [38]. Changes to the underlying tool domain, or modifications to the tools that are being integrated, may necessitate evolution of the representative metamodels and domain models, as discussed next.

2.2.4 Domain Model Evolution
Previous work in the area of domain model evolution focused on the techniques and methods for synthesizing transformations based on changes in the metamodel. Sprinkle’s thesis [39] provides an academic perspective (for the mechanics of synthesizing such transformations, see [40]). Techniques for the graphical specification of the semantics of a modeling language (i.e., the code generator associated with a metamodel) can be found in [41]. A proposal by Bell [42] advocates the creation of a catalog of grammar transformations that are capable of automating the evolution of DSL programs.

3 Formalisms for DSML Evolution
Throughout the rest of the paper, we consider the terms program and model to be equivalent. We consider that a language is either a traditional textual programming language, or a graphical modeling language, since each is eventually represented as an abstract syntax tree. Where appropriate, we utilize notation from Winskel [43], Pierce [44], and Karsai [35]. The purpose of this section is not to lay from the “ground up” a new formalism for specifying DSMLs, but rather a framework in which we can reason about claims of evolution tersely, and with precision.
3.1 Language and Domain

Let $T$ be a (finite) set of types, with $t$ as the element name, and $S$ be a (finite) set of terms used to build a model, with $\tau$ as the element name. We use the symbol $A$ as the abstract syntax of a language (with $a$ as an element), which in this paper we consider as a finite set of constraints on the construction of a potentially infinite set of models.

We abbreviate the abstract syntax tree as AST, and assume it is partially ordered (to guarantee determinicity of the model). We use $A_C$ to refer to the concrete syntax of a language, which specifies the appearance of types and terms during model construction. Outside the scope of our consideration is the process of producing the AST from terms expressed using the concrete syntax.

**Assumption 1 (Semantics-free Concrete Syntax):**

No semantics is interpreted from the $A_C$.

**Definition 1:** Let a domain be defined as $D = (T_D, C_D, S_D)$, where $T_D$ represents a finite set of domain concepts, or types, $C_D$ is a set of constraints applied to relationships between members of $T_D$, and $S_D$ is the well-understood, or perhaps well-defined, set of semantics used to provide meaning to $T_D$ in the domain. Note that a domain is a foundation or motivation for a language, but does not necessarily define an abstract or concrete syntax.

3.2 Attaching Semantics

In this paper, we utilize denotational semantics to express that certain syntax elements are equivalent in meaning, or to defer the evaluation of certain syntax elements by parts, utilizing a sort of late binding to interpret the semantics of the terms of a syntax compositionally. The choice of denotational semantics permits us to talk about how syntax elements are mapped to meanings in other domains, rather than an operational mapping in some abstract machine. This is not a first for describing DSMLs, since many programming language experts have used such semantics before: however, semantics specifications vary widely in DSMLs (graph transformations, code generation, in-editor simulation, etc.). As there is no well-accepted specification for semantics, we choose to represent semantics generally using a denotational formalism.

**Definition 2:** Let the semantics of a DSML, $S$, be defined as a set of partial functions on sets $A$ and $S_D$, formally: $S : (A \rightarrow S_D)$, which can be read as “the set of all partial functions from the abstract syntax to the semantic domain.” Without loss of generality, the denotational semantics of the DSML will be in one of the following sets of partial functions: implicit ($S_I$), explicit ($S_E$), and mixed ($S_M$) domain semantics.

**Definition 3 (Implicit Domain Semantics):** The term implicit domain semantics, $S_I$, indicates that the semantic definition maps from an abstract syntax pattern to a domain type (in $T_D$) with its own semantics. Thus, semantics are not “bound” to this type when the model is given meaning (i.e., at model interpretation time), but the meaning is provided by its behavior in the domain. For example, a translation of a language element Click into a MATLAB/Simulink type Step in the Simulink ‘sources’ library is an implicit domain mapping, and its behavior will depend on its context in the generated model, as well as whichever version of MATLAB is used to run the model. A consequence here (sometimes intended, though not always) is that an upgrade in MATLAB may change the semantics.

**Definition 4 (Explicit Domain Semantics):** Explicit domain semantics emerge when the semantic definition maps to a rewriting of the structure of existing syntax. Thus, semantics are provided during model interpretation time, in the form of additional information (e.g., computable from the structure of the model), or a model of computation in the form of conditional operations or loops. Note that all of these expressions are made in the semantics of the domain, which implies that a subset of the domain semantics, $S_D$, is encoded into the semantics of the DSML. Explicit domain semantics, $S_E$ are explicitly different from $S_I$, which do not specify semantics at all, but delegate semantics to the domain’s (implicit) interpretation of its own types. We permit the application of multiple domain types to be used to satisfy the semantics, rather than a single substitution of $S_I$.

**Definition 5 (Mixed Domain Semantics):**
Semantics are mixed domain semantics when the semantic definition maps to a mixture of implicit, and explicit, domain semantics. Such a mixture could be from repeated transformations of the AST until implicit semantics can be used, or perhaps to rewrite the trees for a different behavior in the semantic domain.

3.3 Constraints

We now address the fact that $S$ is a set of partial functions—meaning that the semantics of certain terms may be undefined, even if their syntax is valid. To limit the creation of such syntaxes, or to inform the model developer that such syntaxes are invalid prior to attaching semantics, constraints are used.
Definition 6 (Constraints): Let constraints be defined as \( C = \{c_1, \ldots\} \), a finite set, where each \( c_i : A_j \to B \) with \( a_j \in A \). Generally speaking, constraints tell whether a valid syntax structure is in fact semantically invalid.

There is a subtle difference between constraints and abstract syntax. Other formalisms (see [35]) include the constraint set as part of the syntactic specification of a language, since both apply to the structural forms of the language. While this is certainly true, abstract syntax better expresses allowed structure, while constraints better express contextually required or un-allowed structure. Thus, for example, we prefer the use of a constraint to express cardinalities such as \( 1..^* \) (especially when only in certain contexts), while abstract syntax expresses cardinalities such as \( 0..1 \). Constraints which require association (or containment) cardinalities of \( 1..^* \) are called constructive constraints in this paper. This subtle distinction will be used to our advantage when proving claims in Section 5.

Some properties that constraints should enforce (or reveal) regarding the attachment of semantics help to understand how syntax and semantics can be engineered. Valid syntax forms, if they are undefined \((\dagger)\) semantically, should provide a constraint function returning false.

\[
(a \in A \land S[[a]] \dagger) \implies \exists c \in C \text{ s.t. } c(a) = \text{false} \quad (1)
\]

3.4 Language Specification

Definition 7 (Metamodel): The metamodel is defined as a model of syntax and constraints, formalized as \( M^* = \langle A_C, A, C \rangle \). Note that we depart from the convention laid forth in [35] to consider the constraints as a part of the \( A \) tuple. This helps to express various subcases of domain evolution in Section 4. The semantic interpretation of \( M^* \) is the synthesis of \( A \) and \( C \) through the definition of types, \( T \), and the allowed relations between members of \( T \) (see [4]).

We occasionally refer to the metamodel as a shorthand for a language without semantics, or the subset of a language without consideration of semantic attachment.

Definition 8 (Language): Let a language, \( \mathcal{L} = \langle M^*, S \rangle \). To maintain generality, we consider that \( S = \{S_1, S_2, \ldots\} \), a finite set. This means that a language can consist of a set of several semantic interpretations, which are not necessarily coupled. Usually, the cardinality of \( S \) is 1, but never 0, implying that a language does not exist without semantics. It is appropriate that many examples in the literature refer to semantic mapping as “model interpretation” given that the models may be interpreted in more than one way. To motivate our explicit mention of multiple interpretation, consider a model of a sensor network, that may be interpreted in one semantics for simulation [45], and by another semantic definition to perform design-space optimization [46]. When the semantics are not defined or immaterial, we use the term metamodel. Some publications in the literature refer to a language as a paradigm (see e.g., [4], [47]).

3.5 Models

Definition 9 (Model): Let \( m \) be a model, an instance of some type, \( t \), in a language, \( \mathcal{L} \). In this paper, we presume that the relationships between any two models obey the abstract syntax, \( A \), of the language.

We use \( M \) to represent a model database, a collection of models according to the definitions of a metamodel, \( M^* \). Although a model database may express typed multigraph relationships, it was shown in [4], [47] that it is possible to encode such relationships into a tree, by maintaining relationship information as attributes. We can, without loss of generality, express that \( M \equiv m_0 \), where \( m_0 \) represents the root model. A model database, \( M_i \), is a partially ordered set (i.e., poset) (since it is expressed in an AST). To express a model’s conformance to various formalisms, we define the following expressions.

Definition 10 (Model Database Conformance): A model database is said to conform to a metamodel, when no constructions violate the syntax rules. We use the relation \( \models : M \times M^* \to B \). Thus, by \( M_1 \models M_0 \), we mean “Model \( M_1 \) conforms to the syntax rules expressed by metamodel \( M_0 \).” Because model conformance is dependent on the syntax defined by a metamodel, we might also say that \( M_1 \models A_0 \), to stress that the \( M_1 \) does not violate any syntax rules defined in \( A_0 \).

Definition 11 (Model Database Well-Formedness): A model database, \( M \), is said to be well-formed in a metamodel, \( M^* \), if \( (M \models M^*) \land (\forall c \in C, m \in M, c(m) = \text{true}) \). We use the relation \( \models \equiv \) as a shorthand, and define it as \( \models : M \times M^* \to B \). Thus, to say that \( M_1 \models M_0 \), we mean that “Model \( M_1 \) is well-formed to the syntax rules and constraints of \( M_0 \).”

By definition, \( M \models M^* \implies M \models M^+ \).

4 DSML Evolution: Impact on Models

Forces outside the control of the language engineer frequently effect change to a DSML, since
changes driven by the domain require changes to the language. After some discussion of these forces and mechanics, we present formalisms of what such changes mean to the existing definitions. Our perspective on the impact of evolution is to address how to ensure that models created according to a metamodel or language conform to, and are well-formed in, the successor of that modeling language. We think this perspective is important, since DSMLs are so easy to modify.

4.1 Driving Forces

For DSMLs, the driving forces of language evolution come from two major perspectives: changes for usability, and changes based on the domain. As we discussed in Section 1, we concentrate on the latter changes. Changes for usability are frequently modifications in the concrete syntax of the modeling language. For visual languages, this may be the icons used to denote certain types in the modeling language, or the colors of certain lines. As long as concrete syntax elements are not used in the specification of semantics (Assumption 1), then their evolution is independent of the languages semantic mapping.

Other language modifications, whether or not for usability, may require changes to the semantics. For example, changes in the ability to specify models using containment or hierarchy may drastically impact the algorithms used to attach semantics. Classic examples can be found in [28]. When semantics change, there are usually—but not always—changes in the abstract syntax as well. Changes in the language will likely mean that models that were well-formed under one language are no longer guaranteed to be well-formed in its successor. That discussion follows an introduction to how domains may evolve.

We can indicate succession of a language, type, term, or any other formalism, e.g., $L'$ indicates that $L'$ is an evolved version of $L$. In this paper, we use the successor notation to indicate that any member of the language tuple has changed.

4.2 Evolution of domain types

Our choice to treat types in a set-theoretic manner yields some trivial observations; nonetheless, a discussion is needed for later claims. Recall that domain types, $\mathcal{T}_D$, are elements of the domain considered familiar to modeler developers. Generally, these are represented in the concrete syntax, $\mathcal{A}_C$, through the use of appropriate images to represent domain concepts, and interfaces (interconnection of domain types) that are similar to those used by domain experts.

There are three possible changes to domain types: (1) Adding a new type(s); and/or (2) Removing an existing type(s); and/or (3) Renaming an existing type(s). These changes are not necessarily mutually exclusive. Regardless of the mechanics of the change, either of these changes produces a new definition of the domain types.

If new domain types are added, then

$$\mathcal{T}_D' = \{t'_{1}, \ldots, t'_{n}\} \cup \mathcal{T}_D$$

where $t_{i} \notin \mathcal{T}_D$ (note $\mathcal{T}_D' \supset \mathcal{T}_D$). If existing domain types are removed, then

$$\mathcal{T}_D' = \mathcal{T}_D \setminus \{t_{1}, \ldots, t_{n}\}$$

where $t_{i} \in \mathcal{T}_D$ (note $\mathcal{T}_D \supset \mathcal{T}_D'$). If one or more existing domain types are renamed, then

$$\mathcal{T}_D' = \{t'_{1}, \ldots, t'_{n}\} \cup \mathcal{T}_D \setminus \{t_{1}, \ldots, t_{n}\}$$

where the expression is evaluated from the right (i.e., types are removed, and then added, to the $\mathcal{T}_D$ set to produce $\mathcal{T}_D'$). In this formalism, we require that $\forall i < n, t_i \rightarrow t_{i}'$, meaning that there is a one-to-one correspondence for replaced types. Since $\mathcal{T}_D' \notin \mathcal{T}_D$, and $\mathcal{T}_D' \notin \mathcal{T}_D$, there is no concise representation for this case. If more than one class of type change occurs, then we see a special case of renaming existing types.

$$\mathcal{T}_D' = \{t'_{1}, \ldots, t'_{n}\} \cup \mathcal{T}_D \setminus \{t_{1}, \ldots, t_{n}\}$$

If $n = m \neq 0$, and $\forall i < m, t_i \rightarrow t_{i}'$, then this type change reduces to that described in (4). If $n = 0$, then this change reduces to that described in (2), and if $m = 0$, this change reduces to that described in (3). If $n = m = 0$, then $\mathcal{T}_D' \equiv \mathcal{T}_D$, and no change in types has occurred.

4.3 Evolution of Domain Semantics

Recall that types in a DSML are anchored to domain semantics using implicit, explicit, and mixed semantic relations. What changes in the domain semantics are possible? Renaming existing types will imply a corresponding renaming of types in the domain semantics. The addition of existing types to the domain will imply a corresponding enhancement of the domain semantics. Further, there may be changes in the semantics which are independent of any changes to the types.

It is outside the scope of this paper to formally express each of these potential changes in semantics, and in many cases will be domain-specific in
the specification of such semantic changes. Generally, though, each of these changes will necessarily require modifications to the language in order to reflect the domain.

4.4 Evolution of Domain Constraints

Changes in the domain constraints may affect a DSML by requiring new constraints—otherwise, it is possible to build models that are not meaningful in the domain. It is also possible that domain constraints will be relaxed, in which case the language may choose to relax the corresponding constraints as well. Without fully specifying these classes of change, they correspond to those of type evolution. Generically, domain constraint evolution is:

$$C'_D = \{c'_1, \ldots, c'_n\} \cup C_D \setminus \{c_1, \ldots, c_n\}$$

where successor constraints removed may not necessarily correspond to successor constraints added.

4.5 Evolution of Language Syntax

The evolution of language syntax, $A$, is motivated chiefly by changes in domain semantics, domain types, and domain constraints. These changes fall into the same categories as domain constraints and domain types, in that adding, deleting, and mixed-mode changes apply. Renaming of elements of an abstract syntax is handled in renaming of types. Thus, the major changes to syntax that are not already covered by type changes are new associations in which types can participate, or a removal of associations in which types may no longer participate. Generically, this is formalized as:

$$A' = \{a'_1, \ldots, a'_m\} \cup A \setminus \{a_1, \ldots, a_n\}$$

If new syntax possibilities are enabled (i.e., new relationships or associations are allowed), then domain models are not affected (i.e., $A \subseteq A' \implies M \models A'$). However, if $A \setminus A' \neq \emptyset$, then if $\exists m \in M$ constructed using $a \in A, a \notin A'$, then this model must be removed, or evolved, according to syntactic changes. These claims follow from the definitions.

4.6 Evolution of Language Constraints

Language constraints are generally changed when domain constraints are modified, added, or removed. This is especially true in DSMLs that heavily utilize implicit domain semantics, since the successor models may violate constraints in the evolved domain. Example motivations for changing language constraints include the removal of semantics from certain constructs, but without explicit removal of syntax (for example, a syntax construct may be permitted, except in the case where a term or model has a certain value). At the root, these changes are motivated by changes in semantics (either in the domain, or the language), and addition of constraints is one way to prevent ill-formed construction of models.

Adding a new constraint may have no effect at all on an existing model, if that model does not currently violate the new constraint. So, if the constraint set grows, and any existing model fails to satisfy a new constraint (i.e., $C' \supseteq C$), and $\exists m \in M, c' \in C' \text{ s.t. } c'(m) = \text{false}$, then the evolution of constraints may necessitate evolution of the model database. If $C' \subseteq C$ and there are no other changes to the language, then $M \models M'^\prime$. This statement follows from the definition of well-formedness.

4.7 Evolution of Language Semantics

Evolution of language semantics is chiefly driven by changes in the domain semantics. Changes to the domain semantics necessarily require changes to the language semantics, if the semantics of the DSML is expected to accurately abstract those of the domain.

Language semantics defined chiefly by implicit domain semantics ($S_I$), generally speaking, require fewer changes to the language semantics during evolution assuming that the desire is to maintain implicit trust in the domain semantics. However, if the goal of evolution is to maintain the modeler’s intent, then semantic equivalence for the successor models is desired, which makes languages defined purely by $S_I$ more difficult to evolve. In this case, language developers may chose to move to a mixed-mode definition for their semantics, perhaps changing the definition of some constructs (currently using implicit semantics) to be explicitly changed prior to their association to domain types.

If a language is defined using explicit domain semantics ($S_E$) then any changes to $S_D$ will require updates to $S_E$ in order to reflect these semantics changes. It may be, then, that no changes will be necessary for any models. Evolution where semantics change, but the metamodel does not, can allow easy “upgrades” for DSML users to the new language, but places the onus on language developers to ensure conformance to $S_D'$.

Evolution of a language that utilizes mixed-mode semantics is complex, and requires special consideration of the changes to the $S_D$. Such evolution
Algorithm: Generic Model Evolution, $\mathcal{X}$  

for all $m \in \mathcal{M}$ do  
  if $\exists \tau \in \mathcal{T}$ such that $f_A(\tau) = \varnothing$ then  
    $m' = f_A(m)$  
  else  
    $m' = m$  
  end if  
end for

Fig. 2. A template algorithm for model evolution, using a syntactic transform set, $\mathcal{F}_A$ (see Equation (9)), on a given model database $\mathcal{M}$. No guarantees are given on $\mathcal{M}'$.  

should be handled on a case-by-case basis, where each change to the domain semantics is analyzed to consider what portions of the language semantics should change to reflect the domains new semantics. The strategy to perform the evolution is a mixture of the above $S_I$ and $S_E$ methods.

4.8 Model Evolution

Model evolution is the transformation of domain models that were created under a language, $\mathcal{L}$, to be well-formed and conform in the successor language, $\mathcal{L}'$. Of particular importance is the question of the semantics of the models under each language. To represent this transformation, we use the relation $\mathcal{X} : \mathcal{M} \to \mathcal{M}'$, and we give a general algorithm to perform model evolution in Figure 2.

We divide these kinds of model evolution tasks into two categories: syntactic model evolution and semantic model evolution.

4.8.1 Syntactic model evolution

In order to define a generic algorithm for syntactic model evolution, we produce some foundational model transformation types.

Definition 12 (Syntactic Transform): Define a syntactic transform, $f_A$, as a partial function that operates on the types of a language. Formally, we express this as:

$$ f_A : \mathcal{T} \to \{ \mathcal{T}' , \varnothing \} $$

where $\mathcal{T} \in \mathcal{L}$, $\mathcal{T}' \in \mathcal{L}'$. We explicitly include $\varnothing$ in the range of $f_A$, because such a transform may map some elements to $\varnothing$, indicating that they are to be removed. This definition of syntactic transform is defined on types, rather than terms, with the understanding that the context and value of instantiated types (i.e., terms) may be used in determining the definition of a transform. Thus, it is appropriate to mention that $f_A$ is defined on terms (e.g., $f_A(\tau)$) as well as models (if those models are terms in a language, e.g., $f_A(m), m \in \mathcal{M}$).

Assumption 2: We assume that a set of syntactic transforms can be partially ordered, to permit deterministic results of the application of a set of $f_A$ on $\mathcal{M}$.

Due to Assumption 2 we rule out the potential that, in Figure 2 e.g., a search for $f_A$ would result in more than one $f_A$ transform for any $m$. Thus, we preclude ambiguity of evolution specifications from our algorithms.

Definition 13 (Syntactic Transform Set): Define a syntactic transform set, $\mathcal{F}_A$, as a partial order of syntactic transform functions. Formally, we express this as:

$$ \mathcal{F}_A = \{ f_{A_1}, \ldots \} $$

a finite set, where (based on Assumption 2) the set of partial functions is partially ordered, i.e., $f_{A_1} \leq f_{A_2} \leq \ldots$.

By requiring this set to be partially ordered, we provide determinicity for a transformation “program” whose steps are determined by the order in which these transforms are applied. Such a program would operate on a model database, and would match the AST of a model database, $\mathcal{M}$, based on the partial ordering of the syntactic transformation specifications.

Such a partial ordering could be:

- emergent (e.g., based on the runtime interpretation of the function signatures, according to the model structure, as in XSL);
- explicit (e.g., based on an ordering specified by the modeler at transformation specification time); or
- mixed.

We leave as implementation details to the transformation designer how the partial ordering is created, but we assume that such a partial ordering exists for this work. Such an assumption is justified, since non-deterministic transformations would, by definition, jeopardize the semantic correctness of the successor model database, and thus be out of the scope of this paper. It is possible to guarantee that a set of transformations will terminate with a deterministic result. For example, the JMangler tool [48] ensures a correct result from a set of transformations by forcing restrictions (termed positive triggering and monotonicity) on the transformations.

Definition 14 (Syntactic Model Evolution): Syntactic model evolution relates a model database, $\mathcal{M} \Rightarrow \mathcal{M}'$, into a successor model database, $\mathcal{M}'$.  


Formally: $\mathcal{X}_A : \mathcal{M} \rightarrow \mathcal{M}'$ s.t. $\mathcal{M}' = \mathcal{M}'^+$ where $\mathcal{X}_A$ is performed as described in Figure 3, using some $\mathcal{F}_A$. Note we assume an isomorphism allowing $m' = m$.

We do not require syntactic model evolution to be an atomic translation, but we instead depend upon the definition of a deterministic syntactic transform set, $\mathcal{F}_A$, to produce a logically atomic translation (though perhaps in several phases which produce intermediate or temporary artifacts). Syntactic model evolution makes no claims of well-formedness for the successor model database (this is discussed in Section 5).

### 4.8.2 Semantic model evolution

**Definition 15 (Semantic Model Evolution):**

Semantic model evolution, $\mathcal{X}_S$, relates a model database, $\mathcal{M} \models \mathcal{M}^+$, into a successor model database, $\mathcal{M}'$.

$$\mathcal{X}_S : \mathcal{M} \rightarrow \mathcal{M}' \text{ s.t. }$$

$$\mathcal{M}' \models \mathcal{M}'^+ \wedge \forall m \in \mathcal{M}, S[m] \equiv S'[m']$$

(10)

Here, we use the notion of *equivalence* between original and successor semantic behavior on the given $\mathcal{M}$; that is, Equation (10) can be read as “the semantics of the successor model, according to the successor semantics, are sufficiently equivalent to those of the original model according to the original semantics.” If multiple semantic interpretations exist, then each member of the set of semantics must be satisfied. Also note that the postcondition requiring well-formedness of the successor model is different from that of syntactic model evolution. We discuss these subtleties in the next section.

In Figure 4 we provide a generic algorithm for performing semantic model evolution.

---

**Algorithm: Semantic Model Evolution, $\mathcal{X}_S$**

**Require:** $\mathcal{M} \models \mathcal{M}^+ \wedge \exists m \in \mathcal{M}, S[m] \equiv S'[m']$

**Ensure:** $\mathcal{M}' \models \mathcal{M}'^+$

for all $m \in \mathcal{M}$ do

if $\neg G_A(m, \mathcal{A}_{M'}(m))$ then

$m' = f_A(m)$

else

$m' = m$

end if

end for

---

**Algorithm: Syntactic Model Evolution, $\mathcal{X}_A$**

**Require:** $(\exists m \in \mathcal{M}' \text{ s.t. } \neg G_A(m, \mathcal{A}_{M'}(m))) \wedge \mathcal{M} \models \mathcal{M}^+$

**Ensure:** $\mathcal{M}' \models \mathcal{M}'^+$

for all $m \in \mathcal{M}$ do

if $\neg G_A(m, \mathcal{A}_{M'}(m))$ then

$m' = f_A(m)$

else

$m' = m$

end if

end for

---

### 5 Analysis/Conjectures of Limitations

As we have described so far, there are various scenarios for evolution, including several magnitudes of evolution (i.e., the number of items in the $\mathcal{L}$ tuple that change). We now discuss several questions that will enable us to determine whether fundamental limitations exist for domain model evolution. At the highest level, we are interested in the following questions:

- What limitations exist, based on the kinds of definitions or semantics used; and
- Do certain classes of evolution imply that semantic transformations are impossible to auto-generate?

Based on the formalisms created previously, we make the following claims:

- Syntax mismatch is not a necessary condition for semantic evolution;
- Proof of semantic evolution is sufficient to prove syntactic evolution;
- Proving syntactic transformation, without regard to semantics or constraints, for an arbitrary model is possible, based on a given model evolution transformation.

These claims require considerable thought to understand intuitively, and prose describing their logical reasoning can be difficult to follow. The formalisms developed in Section 3 allow us to describe and reason about those claims accurately. This section provides such a basis for establishing the proof of our claims, and laying the foundations for future proofs of our conjectures, while concurrently giving some analysis of the results, in preparation for an example.

**Lemma 1:** The need for syntactic model evolution is not a necessary condition for semantic
model evolution. Formally, \( \forall m \in M \) s.t. \( S[m] \neq S'[m] \) \( \rightarrow M \neq M' \).

Proof: Syntactic model evolution requires that the existing model, \( M \), would not conform to the successor metamodel, \( M' \), without some evolution. Imagine a language, \( L \). Let us modify the semantics of this language, and preserve the metamodel. The successor language definition is \( L' = \langle M', S' \rangle \), and \( M' = M'' \). Thus, if \( M = M' \), and \( M' = M'' \), then \( M \equiv M'' \). However, \( \exists m \in M \) s.t. \( S[m] \neq S'[m] \), which is the precondition for semantic model evolution. Thus, semantic model evolution is required, but none of the requirements for syntactic evolution were presented.

This lemma is useful to intuitively understand that there can exist cases where we may not realize that the language has significantly changed, because any models we created in the previous language are well-formed (or at least conform) in the successor language. Importantly, we use this result in discussing semantic model evolution in Section 5.3.

Evidence of semantic evolution is sufficient to prove syntactic evolution: \( M' = \chi_S(M) \implies M' = M'' \). E.g., given \( M' \) as the successor of \( M \), then by Figure 4 \( M' \equiv M'' \). By the definition of well-formedness \( M' \equiv M'' \).

The development of a semantic evolution eliminates the need to consider the issue of syntactic evolution.

5.1 Syntax-Only Model Evolution

Definition 16 (Syntax Checker): Let \( G_A \) determine if syntax is violated. Given the terms of a model \( T \), then \( G_A : T \times A \rightarrow B \). If the relation maps to true, then the structure obeys the syntax rules; otherwise semantics are not necessarily well defined.

Theorem 1: A syntactic model evolution can be constructed for any domain evolution scenario motivated by change in type, or syntax.

Proof: We separate the cases out by the requirements for syntactic evolution, such as changes to the types or syntax, as described in Section 4.2 and Section 4.5.

Trivial Case: \( \forall m \in M \) where \( \neg G_A(m) \), \( M \setminus m \). Thus, all models that do not conform to the new metamodel are removed. This trivial case satisfies our requirements that \( M' = M'' \), though it deletes any model that does not conform, even if modifications to the model would enable conformance.

Type Deletion: By definition, \( T' = T \setminus \{ t_1, t_2, \ldots, t_n \} \). Then, \( \forall m_i \) instanceof \( t_i, M \setminus m_i \). Since all remaining types who appear as models in \( M \) will exist, the model now conforms to \( M'' \).

Type Addition: no changes required to the new model.

Type Replacement: By definition, \( T' = \{ t'_1, \ldots, t'_n \} \cup T \setminus \{ t_1, \ldots, t_n \} \). Then, \( \forall m_i \) instanceof \( t_i \), apply \( t'_i \) to \( m_i \), producing \( m'_i \) instanceof \( t'_i \).

Mixed-mode type changes: By combination of type replacement, and deletion (in that order). Note that the partial ordering of these changes may affect the successor model.

Syntax Changes: for brevity, we point out that the proof is constructive, along the same lines as type changes. For \( A \subseteq A' \), the existing model conforms to the successor metamodel as discussed in their definition in Section 4.5. For \( A' \subseteq A \), the construction in Type Deletion can be used, where all models or relations constructed using syntax elements from \( A \setminus A' \) are removed from the successor model database. For mixed-mode changes, where \( A' \not\subseteq A \land (A \setminus A' \not= \emptyset) \), then the combination of replacement, and deletion (in that order) will remove all non-conforming models from the database, after replaced models have been converted.

5.2 Evolution with Constraints

Constraint satisfaction for a given set of models, after transformation, is possible, under certain conditions in the models and constraints. In this section, we examine how an arbitrary set of models is affected based on enlarged, and reduced, successor constraint sets.

Theorem 2 (Constraint Satisfaction for successor subsets): Show that given \( C \subseteq C' \), and \( M' = \chi(C) \) s.t. \( M' \equiv M \), that \( (M \equiv M'' \land M' \equiv M''') \implies M' \equiv M'' \).

Proof: Assume that \( M \equiv M'' \), \( M' = \chi(C) \), \( M' \equiv M''' \), and \( M' \equiv M''' \). Since \( M' \equiv M''' \), then \( \exists c \in C' \), \( m \in M' \) s.t. \( c(m) = \text{false} \) (so violation of \( \equiv \) must be constraint-based). However, since \( C' \subseteq C \), and \( \exists c \in C' \), \( m \in M' \) s.t. \( c(m) = \text{false} \) (given that \( M \equiv M'' \)), then \( \exists c \in C' \), \( m \in M' \) s.t. \( c(m) = \text{false} \). Now, we have a model, \( M' \) that satisfies all constraints, but is still not well-formed. This contradicts the notion of well-formedness under conformance, since \( M' \equiv M'' \) (by definition of well-formedness).

However, other cases are not so easily proved. This is because the addition of constructive constraints may require new models to be created. To concretize this, we offer the following conjectures:

Conjecture 1: Given \( C' \supset C \), then if \( M \equiv M'' \), or all constraints in \( C \cup C' \) are preventative (i.e., do not require nonzero cardinality of types), or all constraints in \( C' \cup C \) are defined in types or syntaxes not in \( M \), then \( \exists X \) s.t. \( (M \equiv M'' \land M' = \chi(M) \land M' \equiv M''') \).

\[ X \equiv X \]
Conjecture 2: Given \( C' \cap C \neq C \), on an unknown \( M' \), there is no generic algorithm to synthesize \( X \) such that \( \forall M = M' \), \( X(M) = M' \), unless \( C' \subseteq C \) or \( \exists C' \in (C' \cap C) \) s.t. \( C \) is constructive.

Conjecture 2 is subtle, and requires some thought to understand intuitively. The main implication is that constructive constraints are nontrivial to relate to syntax changes in the metamodel. For example, addition of hierarchy to a language does not imply that existing models do not conform—unless there are constructive constraints that require containment cardinalities greater than 0. In this case, is there any straightforward (automated) way to rewrite models to account for the new constraint, and syntax rules, with preservation of semantics? Further discussion of these conjectures is extremely involved, and we leave them to future work.

5.3 Evolution and semantics

The semantic equivalence of evolved models is important for DSML language upgrade, as well as for tool integration. However, as we saw with constraints, any proofs establishing semantic equivalence depend heavily on changes to the semantics.

Theorem 3 (Semantic Equiv., implicit semantics (subset)):

Assume that under \( L \), semantic correctness of language semantics, \( S \), is ensured, and was verified by establishing that all language types are mapped to their corresponding domain types. Given \( M \models M' \), \( S \) defined by \( S_D \), and \( S' \) defined by changes to \( S_D \); it is possible to synthesize a semantic model evolution, \( X_S \), where \( M' = X_S(M) \), and \( S(M) = S'(M') \) if the following hold: \((T' = T) \land (A' \subseteq A) \land (A' \in C) \land (C' \subseteq C)\)

Proof: By construction. Because semantic correctness is established when language types are properly mapped to domain types, and under \( L \), \( S \) is semantically correct, we must (1) describe the synthesis of \( F_A \) such that all types are mapped to domain types, and (2) establish that all models are well-formed (as required by \( X_S \) post-conditions).

Case 1: All \( M' \) instance of \( T', \) mapped to \( T' \). Recall \( T' \in L' \). Now, since \( T' \in T \), \( \exists m \in M \) s.t. \( m \) instance of \( t' \in T' \land t' \neq t \in T \) (i.e., all models are composed of types from the original language). Thus, no model types need to be evolved for conformance.

Since by Assumption 1 concrete syntax carries no semantics, we do not need to consider its changes. Thus, no model types need to be evolved for concrete semantics.

Case 2: Well-formedness of \( M' \). Since \( A' \supseteq A \), then by Theorem 1 no changes are necessary to \( M \) for conformance. By Theorem 2, no changes are necessary for well-formedness.

Some analysis of Theorem 3 is merited. We claim that if we use implicit semantics (recall that implicit semantics maps all language types directly to types in the domain, and presume such types are evaluated using domain semantics) on well-formed models, then a syntactic transform set, \( F_A \), can be synthesized to perform semantic model evolution, \( X_S \), if types, abstract syntax, and concrete syntax have been added (or have not changed), and constraints have been removed (or have not changed).

Conjecture 3: Guarantees of semantic equivalence for transformations on an arbitrary set of models (as constrained by the abstract syntax) is undecidable a priori.

Note that we do not claim that Conjecture 3 makes instance tests for semantic equivalence impossible: only undecidable in the general case. Our claim addresses the synthesis of \( X_S \) in a generic way. Also, the decidability of the transformation is independent from the decidability of the semantic domain (for a note in the domain of hybrid systems, see [49]).

6 Example and Results

As an example, we choose a domain whose semantics are rich, well-understood, but not yet standardized: hybrid systems. The abundance of tools, simulation techniques, modeling formats, and subtle semantic interpretations, provides a rich example for the claims in the previous section.

6.1 Hybrid Systems

A hybrid system is a set of discrete states connected as a graph, where each state maps to a dynamical system, and continuous state information can be used to govern the transition (or not) between discrete states, based on constraints on guard conditions, as well as constraints and invariance properties in the discrete states. This is shown in meta-model form in Figure 5a. Hybrid systems are a useful abstraction for modeling switching controllers, or the behavior of switched systems. In each of the following cases, we consider a hybrid systems modeling language HSML, whose semantics are implicit, and which map to simulation in a tool named Tool. The following sub-cases are not meant to be linearly applied, but instead represent several possible evolutions that might be motivated.
Fig. 5. A collection of graphical metamodels for HSML, for the cases of domain model evolution described in Section 6.2-6.5. The semantics of HSML, $S_{\text{HSML}}$, transforms models to the format required by Tool.

6.2 Model evolution based on types

Consider that Tool decides to rename the State type to Location, namely $t_1' = \text{Location}$, $t_1 = \text{State}$; and that $t_2' \in T_{D_{\text{HSML}}}$, $t_2 \notin T_{D_{\text{HSML}}}$, so $T_{D_{\text{HSML}}} = \{t_1\} \cup T_{D_{\text{HSML}}} \backslash \{t_1\}$. $M_{\text{HSML}}$, reflecting this change in the domain, is shown in Figure 5b. Using type renaming (as in Theorem 1) we can synthesize a syntactic model evolution to perform the renaming of each model. If Tool has not changed its semantics (i.e., if the vendor simply refactored its simulation code to use a new keyword) then our syntactic model evolution also satisfies the requirements for semantic model evolution.

6.3 Model evolution based on syntax

Consider that $S_{D_{\text{HSML}}}$ allows hierarchical composition of States, but that $S_{D_{\text{HSML}}'}$ no longer supports hierarchical composition, since $S_{D_{\text{HSML}}}$ exhibits incorrect behavior using certain hierarchical constructs. $M_{\text{HSML}}'$ is modified by removing abstract syntax elements $a_1, a_2$ (which allowed State containment of State, and State containment of Edge), as shown in Figure 5c. Specifically, $A_{\text{HSML}}' = A_{\text{HSML}} \backslash \{a_1, a_2\}$. Using a well-defined operational semantics, including executing hierarchical hybrid systems (based on work in [50]), a syntactic transformation, $\mathcal{X}_A$, can be created such that states all exist in the same level of hierarchy, and the semantics is equivalent. Alternatively, all objects participating in relationships involving $a_1, a_2$ could be deleted from all $M$ databases, which would enable quick use of $M'$, but at the cost of removing key design objects.

6.4 Model evolution based on constraints

Consider that Tool now disallows certain combinations of hierarchical states, rather than disallowing hierarchy completely, i.e., $C_{D_{\text{HSML}}} = \{c\} \cup C_{D_{\text{HSML}}'}$. To avoid ill-formed models, HSML must be modified, such that $C_{\text{HSML}}' \supset C_{\text{HSML}}$. If any models in existence violated these constraints, they could be rewritten based on transforms motivated by constraint evolution. Importantly, given $M$, if $\exists m \in M, c' \in C$ s.t. $c'(m) \rightarrow \text{false}$, then no evolution is needed (this is true even if the introduced constraint is constructive).

6.5 Model evolution based on semantics

Consider that Tool now uses a different semantics to allow transitions between discrete states. The new semantics are so-called “allowed semantics”, rather than “triggered/as-is” semantics (see [51], [52]). The existing semantics are deterministic, meaning that transitions happen at exact values in the state space (e.g., $x_0$). The successor semantics are non-deterministic, in that transitions are enabled at some value, and required by another value (e.g., a range of $[x_0, x_{\text{max}}]$). Note that $x_{\text{max}}$ is implicit in each state’s Domain attribute.

All existing models will be well-formed in the successor language (and there is no need for syntactic model evolution), but the semantics are likely not equivalent in the successor domain, since transitions are not guaranteed to occur at the lower bound $x_0$. With detailed knowledge of the hybrid systems domain, it is possible to rewrite all transitions to set $x_{\text{max}} = x_0$, and thus to ensure an exact transition. A language whose semantics were previously specified using $S_t$ now utilizes more
\(S_M\) (meaning that some of the domain semantics are encoded during translation to achieve necessary structure to map to a domain type).

An important note in this example is that detailed knowledge of the theory of simulation of hybrid systems is required in order to maintain semantic equivalence. In the event that such knowledge is unavailable—or more pathologically, if Tool has a domain constraint of \(x_0 \neq x_{\text{max}}\)—then semantic equivalence may be more difficult to produce. Nonetheless, it is important to realize that the evolution of the models must be driven by the deep understanding of the semantic domain, if semantic equivalence of evolved models is desired.

As we discussed in Section 5, the construction of pathological cases which make semantic equivalence of the successor models in the successor domain shows that languages whose semantics are defined using implicit domain semantics are at high risk for semantic mismatch when evolving language semantics.

### 6.6 Results

Table 1 summarizes for each class of evolution discussed in this paper how to determine the potential for automated syntactic and semantic evolution of models. The autosynthesis of syntax evolution transforms is consistent across the spectrum of changes to the language (as motivated by changes to the domain). For semantic evolution, the ability to autosynthesize the evolution transforms varies widely. We lump together those cases that are either undecided, or it is unknown that an algorithm exists, as much of these results are left as future work.

<table>
<thead>
<tr>
<th>Evolution Type</th>
<th>(A_T)</th>
<th>(A_S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax/Type Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deleted Type</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Added Type</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Replaced Type</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>New Syntax Patterns</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Removed Syntax Patterns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Removed Constraint</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Added Constraint</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Added Constructive Constraint</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Added Constraint and (M = M')</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Added Preventative Constraint</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Added Constraint using only new types</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantics</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes to (S_D), (L) semantics defined using (S_T, T' = T, A_P, A_C, A' \subseteq A, C' \subseteq C)</td>
<td>(\sim)</td>
<td>A</td>
</tr>
<tr>
<td>Changes to (S_D)</td>
<td>(\sim)</td>
<td>U*</td>
</tr>
</tbody>
</table>

\(\text{A} = \text{Autosynthesis of Evolution transform}\)
\(\text{M} = \text{Autosynthesis depends on specifics of models or metamodels}\)
\(\text{U} = \text{Autosynthesis undecidable, or unknown}\)
\(* = \text{Conjecture}\)

**TABLE 1. Categories of evolution discussed in this paper, and whether syntactic \((A_T)\), and semantic \((A_S)\), transforms can be automatically synthesized for them based on changes to \(L\).**

The ease of developing DSMLs through the specification of metamodels has introduced the need to understand how domain models must evolve to conform to these DSMLs. We have shown that the motivations for understanding this problem stem from the subtleties in how domains, DSMLs, metamodels, constraints, types, and semantics each evolve.

We have shown that for some cases of domain evolution, such as adding types or removing constraints, it is possible or even trivial to evolve models in a straightforward way. In other cases, where types are removed, renamed, constraints are added, or the semantics have changed, then the automated evolution for domain models must be carefully examined. While it may seem trivial to evolve syntactically, it may be impractical to automatically evolve semantically. By using the formalisms we present in this paper, modelers can examine their evolved domain, and models in it, to see whether it qualifies as trivial, or requires expert handling (see Table 1).

DSMLs have matured significantly, and they have the potential to lay the foundations for specification and synthesis of high-confidence software systems. The continued understanding of how models are transformed into the semantic
domain—especially understanding semantic equivalence between domains—can revolutionize how high-confidence systems such as avionics are certified, by allowing evidence-based verification of models and code synthesizers.

ACKNOWLEDGMENTS

The authors thank the participants in the 7th OOPSLA Workshop on Domain-Specific Modeling. Additional thanks are given to the participants of Schloß Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems (07-451) [53].

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and Toyota.

Additional support was provided by NSF CAREER grant CCF-0643725, NSF grant CNS-0930919, and AFOSR award #FA9550-091-0519.

REFERENCES


Jonathan Sprinkle is an Assistant Professor of Electrical and Computer Engineering at the University of Arizona. In 2009, he received the University of Arizona's Ed and Joan Biggers Faculty Development Grant for his work in autonomous systems. Until June 2007, he was the Executive Director of the Center for Hybrid and Embedded Software Systems at UC Berkeley. His research is in the area of intelligent autonomous systems, through building blocks of domain-specific modeling, metamodeling, and generative programming. He was the co-Team Leader of the Sydney-Berkeley Driving Team, a DARPA Urban Challenge collaboration with partners Sydney University, University of Technology, Sydney, and National ICT Australia.

Dr. Sprinkle graduated with the Ph.D. from Vanderbilt University in August 2003, and with his M.S. in August 2000. He graduated with his B.S.E.E. in cursu honorum, cum laude, from Tennessee Technological University in 1999, where he was the first graduate of the Computer Engineering program, and the first Electrical Engineering double major.

Jeff Gray is an Associate Professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB), where he co-directs the Software Composition and Modeling (SoftCom) laboratory. He is an NSF CAREER award winner (2007) and currently serves as the chair of the Alabama IEEE Computer Society. Jeff is a Senior member of IEEE and a member of ACM.

He received a Ph.D. in Computer Science in May of 2002 from the Department of Electrical Engineering and Computer Science at Vanderbilt University in Nashville, Tennessee. While at Vanderbilt, Jeff was a research assistant at the Institute for Software Integrated Systems (ISIS), where he worked on the DARPA Program Composition for Embedded Systems (PCES) project. He also worked on a tool integration project at ISIS that was funded by Boeing. In 1991, he graduated summa cum laude with a BS in Computer Science from West Virginia University in Morgantown, West Virginia.

Marjan Mernik is a Professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science, where he leads the Programming Methodologies Group. He is also an adjunct Professor in the Department of Computer and Information Sciences at University of Alabama at Birmingham. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computation. Dr. Mernik has been in charge of several bilateral projects funded by Slovenian Research Agency such as Slovene-USA project “Grammar Inference Technology and Applications in Software Engineering”, Slovene-Portugal project “Automatic Generation of Language-based Tools” and Slovene-Argentinean project “Implementation of Domain-Specific Languages: Evaluation of Approaches”. Dr. Mernik is a member of the IEEE and the ACM. He received a Ph.D. in Computer Science in 1998 from University of Maribor.