The Power of Rich Syntax for Model-based Development

Ethan K. Jackson, Wolfram Schulte
Microsoft Research,
One Microsoft Way, Redmond, WA
{ejackson,schulte}@microsoft.com

Janos Sztipanovits
Institute for Software Integrated Systems,
Vanderbilt University, Nashville, TN
janos.sztipanovits@vanderbilt.edu

Abstract

During the last century, many general purpose programming languages have been developed, all having rigid syntax and often a von-Neuman view of the world. With the rise of model-based development this changes: Feature-oriented programming, domain specific languages, and platform-based design use rich and custom syntaxes to capture domain specific abstractions, refinement mappings, and design spaces. In this paper we show how a formalization of rich syntax can be used to compose abstractions, validate refinement maps, and construct design spaces. We describe a tool FORMULA *for computing these properties, and present a series of examples from automotive embedded systems.*

1. Introduction

Research in model-based development has produced new approaches to architecting and modularizing software systems. Promising approaches include: model-driven architecture [45], platform-based design [54], domain-specific languages [29], feature-oriented design [51] and aspect-oriented programming [30]. Though many of these approaches overlap [34], each offers unique strengths from a unique perspective. However, all of these perspectives makes it unclear how to compare, reuse, and generalize accomplishments in model-based development.

We observe that a key commonality exists across the spectrum of approaches: Design artifacts are captured using (1) rich and customizable syntactic constructs and (2) expressive constraints over syntax. By *rich syntactic constructs* we mean notations that are (at least) set-based and relational, e.g. graph-theoretic [18]. By *expressive constraints* we mean well-formedness rules limiting syntax via type-constraints and context-dependent invariants, e.g. *OCL* [46]. We use the term *rich syntax* as a loose shorthand for these properties. Unlike traditional programming languages where syntax is fixed for all problems, model-based approaches encourage rich custom syntaxes to be created on a *per problem* basis. This new dimension of expressiveness allows engineers to capture design invariants and abstractions using custom expressive notations.

In this paper we provide a novel framework for rich syntax arising in model-based development. We define:

- *domains* as a formalization of rich syntax, utilizing extended Horn logic to represent the constructs and invariants of modeling artifacts (Section 3).
- *composition operators* for building new domains from existing ones. We also check that compositions do not contain contradictions (Section 4).
- *transformations* as translation procedures between domains. We check that transformations are well-behaved, i.e. well-formed inputs are always translated to well-formed outputs (Section 5).
- *design spaces* as sets of syntactic instances over domains. Design spaces can be compactly represented and elements can be enumerated (Section 6).

We have developed a tool called FORMULA that implements these tasks. This implementation uses a *model finding* procedure to construct sets of rich syntax satisfying key properties [24]. The examples in this paper are written in the notation of FORMULA.

2. Rich Syntax in Model-based Development

The automotive community has been an important earlier adopter of modeling technology[52, 61]. Many modeling approaches converge in the automotive domain, making it an ideal place to study modeling foundations. In this section we introduce three automotive examples, each of which is constructed using a different modeling technique. Along the way, we show that *rich syntax* spans all of these modeling styles, providing a common substrate for composition and analysis. This overview focuses on rich syntax, and does not attempt to be exhaustive in any sense.

2.1 Features and Aspects

Feature-oriented design [51, 5] and *aspect-oriented programming* [30] are attempts to modularize and reuse software components even when the components have complex coupling between them. Feature-oriented design partitions a system into *features*, where each feature represents some slice of the overall system. Distinct features utilize overlapping parts of the implementation (e.g. classes, components, etc...) and thus have coupling between them. The goal of feature-oriented design is to mix and match features to create variants (product lines) of a large software system. This vision requires mechanisms to reason about which sets of features are compatible.

A *feature diagram* abstracts the coupling between features allowing the engineer to reason about the possible system variants. Figure 1 shows some features in a car. The diagram describes a tree of features and the interactions between features. For example, the car feature must have a maneuvering feature, but cruise control is optional. The acceleration feature requires exactly one of the manual or automatic features. If cruise control is in the car then the automatic feature must be selected.

Let F be a set of features, then a legal program variant $V \subseteq F$ is a subset of F satisfying the feature diagram. It was observed in [6] that each feature diagram induces a BNF grammar, and this provides a formal basis for checking if a program variant V is legal. Table 1 shows a partial grammar induced by the car feature diagram. Leaf nodes in the diagram become terminal tokens in the grammar (written in upper-case); internal nodes are non-terminals (written in lower-case).

More complex forms of coupling are difficult to capture if feature diagrams are formalized as BNF grammars. For example, the implication from Cruise Control to Automatic transmission is problematic. Consequently, various extensions and formalizations have been pursued [13]. For instance, in [36] feature diagrams are reduced to first-order propositional formulas permitting Boolean constraints between features. This is an exemplar of rich syntax in model-based development.

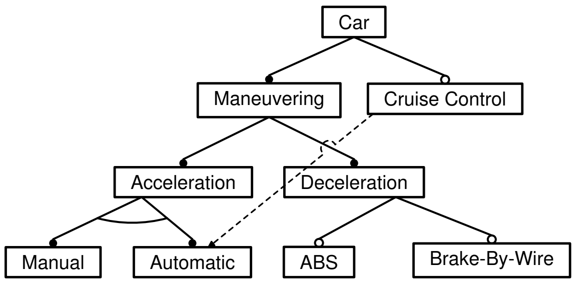


Figure 1. Feature diagram of a car's subsystems.

car:	maneuvering [CRUISE CONTROL]
maneuvering:	acceleration deceleration
acceleration:	MANUAL | AUTOMATIC
deceleration:	[ABS] [BREAK-BY-WIRE]

Table 1. Feature diagram as a grammar.

2.2 Domain-specific Languages

Domain-specific languages (DSLs) evolved from the observation that software development is application-specific and the application context imposes strong constraints on the structure of software [26, 55]. DSLs capture domain-specific concepts, and come in two parts: The *abstract syntax*, which is a rich syntax, and the *behavioral semantics*, which formally assigns behaviors to instances of the abstract syntax. The behavioral semantics is often expressed using a mathematical description of behaviors and can also be domain-specific [11].

For example, in the automotive domain software executes on embedded processors called *ECUs* (electronic control units). ECUs communicate with each other through local-area networks, called *buses*, that have strict rules on the structure, duration, and frequency of *messages* sent between ECUs. Any software system must take such application-specific constraints into account. Figure 2 shows a *metamodel* describing the abstract syntax for an automotive DSL. The diagram is an extended type of *UML diagram*, called a MOF (meta-object facility) metamodel [44]. It explains that a Car is a basic concept parameterized by data indicating the make and model. Every Car contains one or more ECU entities, and each ECU may contain Message entities. A Car contains Buses that link ECUs together. Buses come in two varieties: CAN and FlexRay [56]. Additionally, metamodels can be annotated with *constraints*, often written in the *object constraint language* (OCL). For example, we may add the OCL-like constraint:

$$\forall\, \textsf{ECU}\; e,\ \forall\, \textsf{Bus}\; b,\ (e.isCritical \wedge b.dst = e) \Rightarrow b\ \textsf{is}\ \textsf{CAN}$$

i.e. every ECU containing a safety-critical task communicates on a CAN bus. Loosely speaking, an instance of the abstract syntax is a set of entities that conform to the metamodel, including its constraints.

This simple example shows that rich syntaxes are prevalent in DSLs. At first glance, DSLs resemble graph-theoretic objects. For example, ECUs are vertex-like and Buses are edge-like. This has lead to significant work in extending graph-grammars to capture the rich syntax of DSLs

[7, 48]. These extensions must address data-types, containment hierarchies, non-binary relations, and hierarchy-crossing constructs.

2.3 Platform-based Design and Model-Driven Architecture

Platform-based design and *Model-Driven Architecture* (MDA) focus on migrating abstract specifications to implementations through incremental translation steps [45, 54]. The engineer begins by developing a *functional model* of the system, also called a *platform-independent model* (PIM) in MDA. The functional model is an abstract description of *what the system should do*. Along side the functional model is an *architectural model* specifying *what the system can do*. The goal of platform-based design is to find an appropriate *platform mapping* (translation) from the function to the architecture so that the system is correctly implemented [50].

The left-hand side of Figure 3 shows a functional model, architectural model, and platform mapping. The functional model consists of three tasks (gray circles) S,T, and U. The undirected edges between the tasks are resource constraints. A pair of tasks with resource constraints cannot execute on the same processor, because local resources (e.g. memory capacity) are insufficient to support both tasks. The architectural model consists of three processors (squares) P1, P2, and P3 indicating the processing capability of the implementation. The platform mapping must place tasks onto processors without violating resource constraints. The figure shows one such mapping. In this example finding a platform mapping is equivalent to the NP-hard coloring problem. This is typical in platform-based design, and is a manifestation of the difficulty in changing abstraction levels. (This example is adapted from the problem of scheduling tasks with *conflict graphs* over multiple processors [38].)

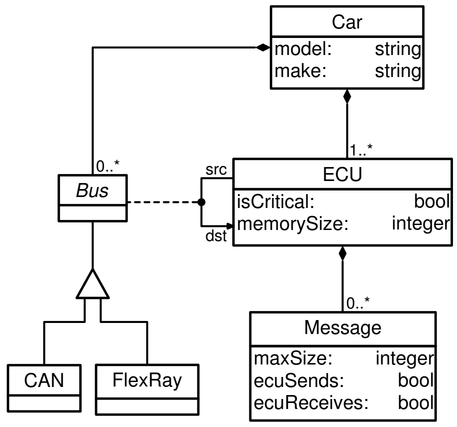


Figure 2. Metamodel of ECU/bus architecture.

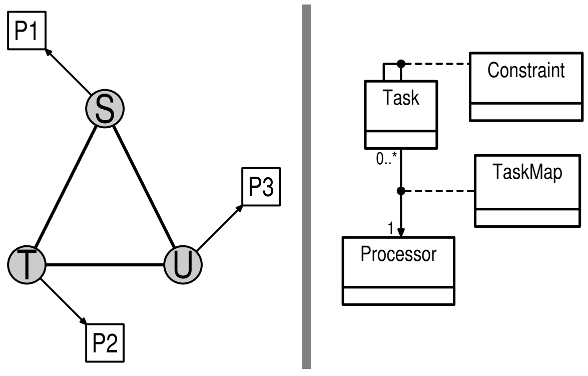


Figure 3. (Left) Function, architecture, and platform mapping, (Right) Metamodel of problem domain.

Rich syntax also plays a key role in platform-based design. It is used to describe the functional and architectural abstraction layers, and to define the valid platform mappings. The right-hand side of Figure 3 shows the metamodel defining exactly this problem domain. According to this metamodel, Tasks are connected by resource Constraints and each Task is mapped to exactly one Processor. This metamodel also has a constraint (not shown in figure) that two tasks connected by Constraint edges cannot be mapped to the same Processor. This well-formedness rule is expressed within the rich syntax, and does not require any knowledge of the computations performed by tasks and processors. The set of all instances conforming to this metamodel is exactly the set of possible functions/architectures with valid mappings.

The observation that the legal syntactic instances correspond to behaviorally meaningful designs has lead to new techniques for *design space exploration* [8]. For example, given a set of tasks T with resource constraints, then there is an associated set of architectures that admit valid platform mappings. This design space of legal architectures/mappings can be characterized once the rich syntax is formalized. Design-space exploration using rich syntax has been realized for specific problem domains [43, 27].

2.4 Rich Syntaxes for Composition

We have shown that rich syntaxes span model-based development. More importantly, rich syntax can be used to compose modeling approaches. Figure 4 illustrates how development styles realistically interact through rich syntax. This figure shows an instance of Car from the DSL presented in Figure 2. The squares labeled $E_{1,2,...}$ are instances of ECUs connected by Buses. (Messages and other data are not shown.) The network of ECUs and Buses forms

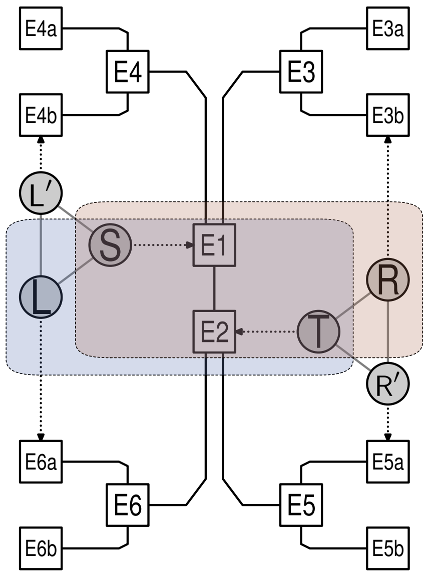


Figure 4. Composition of modeling styles through rich syntax.

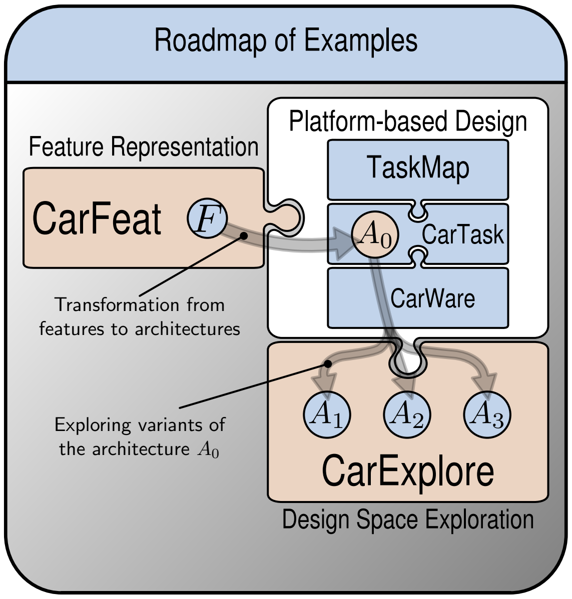


Figure 5. Overview of examples

an architectural model, in the sense of platform-based design. Any functionally correct software system must respect this architecture. The arrows show the platform mapping of Tasks to Processors, where the Processors are actually ECUs.

The shaded areas show subfeatures of the Cruise Control feature. The tasks L, R correspond to sensors on the left/right side of the vehicle. The task S calculates engine dynamics, while T is the main controller used by the cruise control. The two subfeatures span the implementation and overlap at their intersection: $\{\mathsf{S}, \mathsf{T}, \mathsf{E}_1, \mathsf{E}_2\}$.

Though composition of modeling styles seems natural, there are theoretical issues that must be resolved. First, consider the "replacement" of Processors with ECUs. This replacement binds the two rich syntaxes in a non-trivial way: ECUs inherit constraints on Processors and vice-versa. In general, this composition may be ill-defined, i.e. no legal instance exists in the composition. Second, syntaxes can be related in more complex ways than replacement, i.e. by transforming from one syntax to another. However, transformations may themselves contain mistakes in the form of nonsensical rewrites. For example, we must ensure that any transformation process always produces well-scheduled Tasks. Third, modeling provides a means to explore architectures before implementation. For instance, the possible schedules of Tasks to ECUs represents a large space of architectural variants. The challenge is to compactly represent and explore these design choices.

2.5 Roadmap of Examples

In the remainder of this paper we develop the three preceding examples, initially given in isolation, using a unified framework enabling early end-to-end analysis. These examples are expressed in the language of FORMULA (**For**mal **M**odeling **U**sing **L**ogic **A**nalysis). FORMULA is a new tool using Horn logic extended with stratified negation as a basis for integrating and analyzing rich syntaxes and transformations occurring in model-based development [24]. Figure 5 shows the parts that we develop and their interrelationships. We begin by developing a rich syntax for *platform-based design of automotive systems*, as shown in the figure by the block labeled Platform-based Design. First, we define a generic platform devoid of any automotive-specific concepts. This is formalized with the TaskMap domain. Next, we construct a domain, called CarWare, for automotive-specific hardware architectures that does not support platform-based design. Third, these two domains are composed to form the CarTask domain, which can be used to perform automotive-specific platform-based design. This example shows how composition and theorem proving can be used to incrementally build domains with known properties.

The second example builds a feature language for automotive systems. (See the CarFeat block in Figure 5.) The CarFeat domain captures the grammar of the feature diagram in Figure 1. Next, a transformation (syntax translator) called FeatMap is defined, which converts feature sets into partial architectures. In the figure a feature set F it translated into a partial architecture A_0. This example shows how to construct transformations and prove that certain errors are absent from the transformations.

The third example uses FORMULA for design space exploration. We define the CarExplore domain as a subset of the legal CarTask instances. CarExplore instances correspond to a set of interesting architectural variants for automotive embedded systems. All of these examples are linked together to perform design space exploration for a particular feature set. In the figure A_1, A_2, and A_3 are architectural variants of A_0 that were calculated by FORMULA. This realizes the vision of Figure 4, which integrates many different modeling approaches in a consistent manner.

3 Domains and Model Generation

3.1 Elements of Rich Syntax

We have shown that rich syntaxes are built from datatypes, relations over sets/relations, and expressive context-sensitive constraints. We formalize rich syntax by:

1. Using *terms* (i.e. uninterpreted function symbols and typed constants) to encode problem-specific sets and relations,
2. Capturing language invariants through logic programming,
3. Building new syntaxes through formal first-class composition operators.

We illustrate this approach by formalizing the rich syntax of the task mapping language (Figure 3).

3.1.1 Signatures and Terms

A function symbol, e.g. $f(\cdot)$, is a symbol denoting a unary function over a universe U. We say that f stands for an *uninterpreted function* if f satisfies no additional equalities other than $\forall x \in U,\ f(x) = f(x)$. Let Σ be an infinite alphabet of constants, then a *term* is either a constant or an application of some uninterpreted function to a term. For example, $\{1, f(2), f(f(3))\}$ are all terms assuming $\{1,2,3\} \subseteq \Sigma$. Henceforth, our function symbols will be n-ary to capture relations and other constructs. Constructing terms generalizes for arbitrary arity.

Uninterpreted functions form a flexible mechanism for capturing sets, relations, and relations over relations without assigning any deeper interpretations (behavioral semantics) to the syntax. A *finite signature* Υ is a finite set of n-ary function symbols. The *term algebra* $\mathcal{T}_\Upsilon(\Sigma)$ is an algebra where all symbols of Υ stand for uninterpreted functions. The universe of the term algebra is inductively defined as the set of all terms that can be constructed from Σ and Υ.

```
1.  model SchedExample : Taskmap {
2.    /// Tasks and Processors
3.    t1= Task("S"), t2= Task("T"),
4.    t3= Task("U"), p1= Processor("P1"),
5.    p2= Processor("P2"),
6.    p3= Processor("P3"),
      /// Constraints
8.    Constraint(t1,t2), Constraint(t2,t3),
9.    Constraint(t3,t1),
      /// Task mappings
11.   Taskmap(t1,p1), Taskmap(t2,p2),
12.   Taskmap(t3,p3)
13. }
```

Figure 6. Task scheduling instance as a set of terms.

It is standard to let $\mathcal{T}_\Upsilon(\Sigma)$ either denote the term algebra or its universe.

We view a rich syntax as providing a term algebra whose function symbols characterize the key sets and relations through uninterpreted functions. For example, the task mapping language can be encoded using the following signature:

$$\Upsilon_{\mathit{Taskmap}} = \left\{ \begin{array}{c} \mathit{Task}(\cdot), \mathit{Processor}(\cdot), \\ \mathit{Constraint}(\cdot,\cdot), \mathit{Taskmap}(\cdot,\cdot) \end{array} \right\} \quad (1)$$

The $\mathit{Constraint}(\cdot,\cdot)$ symbol is used to encode a binary relation over tasks, while the $\mathit{Taskmap}(\cdot,\cdot)$ symbol is a relation over tasks and processor. (We assume a countably infinite alphabet Σ.)

A *syntactic instance* of some rich syntax is a finite set of terms over its term algebra $\mathcal{T}_\Upsilon(\Sigma)$. The set of all syntactic instances is then the powerset of its term algebra: $\mathcal{P}(\mathcal{T}_\Upsilon(\Sigma))$. For example, Figure 6 shows the task scheduling instance of Figure 3 encoded as a syntactic instance with FORMULA. The keyword model (Line 1) denotes a syntactic instance, the contents of which are terms. This declaration also names the instance as SchedExample, and identifies the specification that contains the associated term algebra. (We describe the Taskmap specification shortly.) Lines 1-6 instantiate the three Tasks and three Processors. The notation:

t1 = Task(“S”)

allows the identifier t1 to stand for the term Task(“U”) where “U” is a string constant. Therefore, Line 8 includes the term:

Constraint(Task(“S”),Task(“T”))

after expanding the identifiers t1 and t2.

3.1.2 Terms With Types

The powerset of terms contains many unintended instances of the syntax. For example, the term

Taskmap(Constraint(Task("S"),Task("T")), Task("U"))

never belongs to a meaningful task mapping instance, because Taskmaps are not intended to relate Constraint terms with Task terms. Erroneous terms can be eliminated by *typing* the arguments of uninterpreted functions:

Taskmap : (Task, Processor).

This statement declares a binary function Taskmap where the first argument must be a Task term and the second must be a Processor term. The function is undefined when applied to badly-typed values, otherwise it behaves exactly like an uninterpreted function.

This enrichment of the term algebra semantics with types leads naturally to an *order-sorted* type system. We formalize this type system now. An *order-sorted alphabet* $\Sigma_{\subseteq}$ is a structure:

$$\Sigma_{\subseteq} = \langle I, \preceq, (\Sigma_i)_{i \in I} \rangle \tag{2}$$

The set I, called the *index set*, is a set of sort names (alphabet names). Associated with each sort name $i \in I$ is a set of constants Σ_i called the *carrier* of i. An order sorted alphabet has the following properties:

$$\Sigma = \bigcup_{i \in I} \Sigma_i, \quad (i \preceq j) \Leftrightarrow (\Sigma_i \subseteq \Sigma_j) \tag{3}$$

In other words, Σ is the union of smaller alphabets and alphabets are ordered by set inclusion; the sub-typing relation $\preceq$ is set inclusion. A *type* τ is a term constructed from function symbols and elements of I or the special top type $\top$. Each type τ identifies a subset $[\![\tau]\!] \subseteq \mathcal{T}_{\Upsilon}(\Sigma)$ according to:

1. The top type is the entire term algebra:

$$[\![\top]\!] = \mathcal{T}_{\Upsilon}(\Sigma) \tag{4}$$

2. A sort name $\tau \in I$ is just the carrier set Σ_τ:

$$\forall \tau \in I, \ [\![\tau]\!] = \Sigma_\tau \tag{5}$$

3. Otherwise $\tau = f(\tau_1, \tau_2, \ldots, \tau_n)$ where f is an n-ary function symbol:

$$[\![\tau]\!] = \left\{ v \in \mathcal{T}_{\Upsilon}(\Sigma) \;\middle|\; \begin{array}{c} v = f(v_1, v_2, \ldots, v_n) \wedge \\ \bigwedge_{1 \leq j \leq n} v_j \in [\![\tau_j]\!] \end{array} \right\} \tag{6}$$

The sub-typing relation $\preceq$ is extended to arbitrary types:

$$\forall \tau_p, \tau_q \quad (\tau_p \preceq \tau_q) \Leftrightarrow ([\![\tau_p]\!] \subseteq [\![\tau_q]\!]) \tag{7}$$

We now apply this type system to the specification of the Taskmap abstraction, as shown in Figure 7. Specifications of abstraction layers are called *domains* in FORMULA; Line 2 declares the Taskmap domain. Lines 4-7 declare uninterpreted function symbols with types. For example, Line 4 introduces a function for instantiating tasks, which has a single argument that is the name of the task being instantiated. The exact data format of a task's name is unimportant, so it is given the top type ($\top$) with the keyword Any. For convenience function arguments can be labeled, but these labels do not affect the typing rules. The syntax:

Task : (name : Any).

assigns the label name to the single argument. Lines 6-7 require the Constraint and Taskmap functions to accept only Task and/or Processor terms as arguments. The remainder of the specification is described in the next section.

3.1.3 Expressive Constraints with Logic Programming

Rich syntaxes often contain complex rules on the construction of syntactic instances; these rules cannot be captured by simple type-systems. One common solution to this problem is to provide an additional constraint language for expressing syntactic rules (e.g. OCL). Unlike other approaches, we choose logic programming (LP) to represent syntactic constraints because:

1. LP extends our term-algebra semantics while supporting declarative rules,

2. The fragment of LP supported by FORMULA is equivalent to full first-order logic over term algebras thereby providing expressiveness [14],

3. Unlike purely algebraic specifications, there is a clear execution semantics for logic programs making it possible to specify model transformations in the same framework

4. Many analysis techniques are known for logic programs; we have adapted these to analyze FORMULA specifications [24].

FORMULA supports a class of logic programs with the following properties: (1) Expressions may contain uninterpreted function symbols, (2) The semantics for negation is *negation as finite failure*, (3) All logic programs must be *non-recursive* and *stratified*. We summarize this class now.

```
/// Platform mapping abstraction
domain Taskmap {
  /// Essential concepts
  Task       : ( name : Any ).
  Processor  : ( name : Any ).
  Constraint : (Task,Task).
  Taskmap    : (Task,Processor).
  /// Key Mapping Constraints
  noMap :? t is Task, fail Taskmap(t,_).
  badMap :? Taskmap(s,p), Taskmap(t,p),
            Constraint(s,t).
  /// Shorthand for endpoints
  Task(x) :- Constraint(Task(x), y).
  Task(y) :- Constraint(x, Task(y)).
  Task(x) :- Taskmap(Task(x), y).
  Processor(y) :- Taskmap(x,
                       Processor(y)).
  /// Model conformance
  conforms :? !noMap & !badMap.
}
```

Figure 7. FORMULA **description of platform-mapping problem**

Definitions. Let V be a countably infinite alphabet of *variables* disjoint from basic constants: $V \cap \Sigma = \emptyset$. Let the term algebra $\mathcal{T}_v$ be an extension of a term algebra with these variables: $\mathcal{T}_\Upsilon(\Sigma \cup V)$. For simplification, we write $\mathcal{T}_g$ for $\mathcal{T}_\Upsilon(\Sigma)$. A term t is called a *ground term* if it does not contain any variables; $\mathcal{T}_g$ is the set of all ground terms. A *substitution* ϕ replaces variables with ground terms. Formally, ϕ is a homomorphism from terms (with variables) to ground terms that fixes constants. We write $\phi(t)$ for the ground term formed by replacing every variable x in t with $\phi(x)$. Two terms s and t *unify* if there exists a substitution ϕ such that $\phi(s) = \phi(t)$.

Expressions. Logic programs are built from expressions of the form:

$$h \leftarrow t_1, t_2, \ldots, t_n, \quad \neg s_1, \neg s_2, \ldots, \neg s_m.$$

where $h \in \mathcal{T}_v$ is a term called the *head*. The sets $\{t_1, \ldots, t_n\} \subseteq \mathcal{T}_v$ and $\{s_1, \ldots, s_m\} \subseteq \mathcal{T}_v$ are sets of terms collectively called the *body*. Each t_i is called a *positive term* and each s_j is called a *negative term*. In the k^{th} expression an implicit relation symbol R_k surrounds each body term; an implicit relation symbol R'_k surrounds the head term h.

$$R'_k(h) \leftarrow \begin{array}{l} R_k(t_1), R_k(t_2), \ldots, R_k(t_n), \\ \neg R_k(s_1), \neg R_k(s_2), \ldots, \neg R_k(s_m) \end{array}.$$

Intuitively, this expression means the following: If there exists a substitution ϕ so that each $\phi(t_i)$ is in relation R_k and for each s_j there is no substitution ϕ' that places $\phi'(s_j)$ in R_k, then add $\phi(h)$ to the relation R'_k [1].

The semantics of logic programming languages vary on how this intuition is formalized. The formalization must take into account the generality of allowed expressions and the mechanism by which the implicit relations are calculated. The fragment we utilize is a well-behaved fragment called *non-recursive* and *stratified* logic programming with *negation as finite failure*. Logic programs of this fragment always terminate and have expressive power equivalent to first order logic.

Semantics. Let $\prec$ be a relation on expressions. Expressions e_i and e_j are related ($e_i \prec e_j$) if the head h of e_i unifies with some term in the body of expression e_j, regardless of whether the body term is positive or negative. A finite collection of expressions $(e_i)_{i \in E}$ is non-recursive and stratified if $\prec$ is an acyclic relation.

Let $o : E \rightarrow \mathbb{Z}^+ \cup \{0\}$ be an ordering of non-recursive and stratified expressions that respects $\prec$:

$$\forall i, j, \in E, \ (e_i \prec e_j) \Rightarrow o(i) < o(j). \tag{8}$$

Using this ordering, the k^{th} expression tests for the presence and absence of body terms in a relation $R_{o(k)}$. Whenever these tests succeed for some substitution ϕ, then the substituted head term $\phi(h)$ is added to the relation $R_{o(k)+1}$.

$$R_{o(k)+1}(h) \leftarrow R_{o(k)}(t_1), \ldots, \neg R_{o(k)}(s_1), \ldots \tag{9}$$

This rule is used in conjunction with the general rule: Whatever can be found in relation R_i can also be found in relation R_{i+1}.

$$\forall i \geq 0, (t \in R_i) \Rightarrow (t \in R_{i+1}). \tag{10}$$

Additionally, LP uses the *closed world assumption*: $t \in R_i$ if and only if it is in R_0 or it is placed in R_i by the application of rules (9) or (10). The *input* to a logic program is the initial relation R_0. The program executes by working from smallest-to-largest expressions, as ordered by o, building the contents of the relations along the way. Note that for non-recursive and stratified programs the choice of o does not affect the results of the logic program.

Negation as Failure. *Negation as failure* (NAF) allows an expression e_k to test for the absence of some terms in the relation $R_{o(k)}$. Developing a semantics for NAF in arbitrary logic programs has been one of the biggest challenges for the LP community. Fortunately NAF is well-behaved in our fragment. The only question is how to interpret variables appearing in negative terms. For example:

$$f(x) \leftarrow g(x), \neg h(x, y). \tag{11}$$

We use the interpretation consistent with the SLDNF-resolution procedure found in standard PROLOG: The variable y is effectively a wild card, so the body succeeds if there exists a substitution ϕ such that $\phi(g(x)) \in R_{o(k)}$ and for all other substitutions ϕ', $\phi'(\phi(h(x,y)) \notin R_{o(k)}$. Substitutions fix constants, so this is equivalent to the condition that: $\exists \phi, \forall \phi',\ f(\phi(x)) \in R_{o(k)} \wedge h(\phi(x), \phi'(y)) \notin R_{o(k)}$. More generally, let P be the set of all positive terms and N be the set of all negative terms in an expression. We write $\phi(T)$ for the application of a substitution to each term $t \in T$. The body of an expression succeeds if:

$$\exists \phi, \forall \phi',\ \phi(P) \subseteq R_{o(k)} \wedge \phi'(\phi(N)) \cap R_{o(k)} = \emptyset. \quad (12)$$

We are interested in finite relations R_k, so we require that the variables in the head term h appear in some term positive term $t \in P$.

Queries. A query is just an expression that does not add new terms to any relations. In FORMULA each query q has a name, which is a boolean variable that is true whenever the body of the query is satisfied, otherwise it is false:

$$qname\ :?\ t_1, t_2, \ldots, t_n, \quad \neg s_1, \neg s_2, \ldots, \neg s_n. \quad (13)$$

Queries are identified by the special operator ":?". Since queries do not modify any relations, it is never the case that $q_i \prec e_j$. Otherwise, queries are ordered like clauses for the purpose of execution. As a convenience, queries can be composed with standard boolean operators into new queries, but this is only syntactic sugar.

Domain Constraints. We use logic programming to capture the complex rules of rich syntaxes. This is done by writing a special query called conforms, which evaluates to true exactly when a syntactic instance satisfies all the rules. The process for testing if an syntactic instance X conforms to the syntax is:

1. Set $R_0 = X$.
2. Run the logic program.
3. Check if conforms evaluates to true.

Lines 9-10 of Figure 7 show the key subqueries for conformance testing of a Taskmap instance. The noMap query tests for Task instances that have not been mapped to Processors. The FORMULA notation "t is Task" is shorthand for adding the positive body term Task(x) and declaring a local identifier t which stands for this term. The keyword fail means negation as failure, so fail Taskmap(t,_) tests that there is no appropriate mapping to any Processor. The underscore character is shorthand for a new variable that does not occur anywhere else in the expression; it emphasizes that this variable behaves like a wild-card.

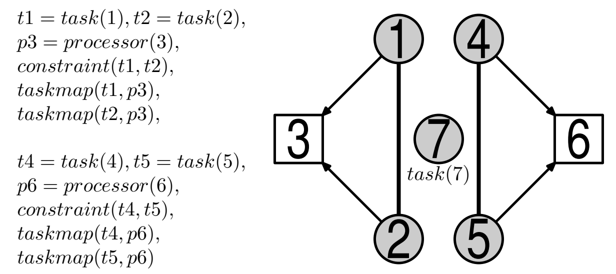


Figure 8. Syntactic instance with bad task assignments and an unmapped task.

The badMap query checks for badly-scheduled tasks. Line 19 builds the conforms query from a boolean combination of noMap and badMap. The expressions in Lines 13-17 allow the logic program to assume the existence of Tasks and Processors terms appearing in Constraint and Taskmap terms. Figure 8 shows a syntactic instance violating both of these rules. Task(7) satisfies the noMap query, while the assignments $(s = 1, t = 2, p = 3)$ and $(s = 4, t = 5, p = 6)$ satisfy the badMap query. However, the instance in Figure 6 does conform to the domain.

3.2 Domains

We have used the term *domain* to describe a constraint system, defined through logic programming, which captures the rich syntax of an abstraction layer. In this section we formalize the concept of a domain. A *domain* D is a structure consisting of: (1) a finite signature Υ_P of *primitive* uninterpreted functions symbols (with typed arguments), (2) a finite signature Υ_R of *derived* uninterpreted function symbols and, (3) a set of logic programming expressions E, one of which must be the special query conforms.

$$D = \langle \Upsilon_P,\ \Upsilon_R,\ E \rangle \quad (14)$$

We assume that a fixed order-sorted alphabet $\Sigma_\subseteq$ is common to all domains. The additional signature Υ_R contains function symbols used for the intermediate storage of data during execution of the logic program. We call these *derived* symbols, because they are always derived from an input instance X and never appear directly in any syntactic instance. In FORMULA derived symbols are not explicitly declared and always start with a lower-case letter. *Primitive* symbols (Υ_P) may appear in syntactic instances; they start with an upper-case letter and must be explicitly declared. All the examples until this point have used only primitive symbols.

Previously we showed that domains can be automatically extracted from metamodels and other modeling artifacts [25]. In this paper we do not focus on converting

```
/// Partial architecture
resc :? Constraint(x,y), Constraint(y,z).
/// Finish the design for me...
threePath :? resc & conforms.
```

Figure 9. Queries for describing a design space.

$t1 = task(1), t2 = task(2), t3 = task(3)$
$constraint(t1, t2),$
$constraint(t2, t3),$

$p4 = processor(4), p5 = processor(5),$
$taskmap(t1, p4),$
$taskmap(t2, p5),$
$taskmap(t3, p4),$

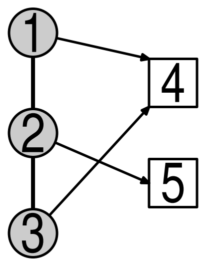


Figure 10. Syntactic instance generated by FORMULA

```
   /// Restriction to finite models
2. domain ThreeSpace restricts Taskmap {
3.   bounds {
4.     t1=Task(x), t2=Task(y), t3=Task(z),
5.     p1=Processor(u), p2=Processor(v),
6.     p3=Processor(w), Constraint(t1,t2),
7.     Constraint(t2,t3), Taskmap(t1,p1),
8.     Taskmap(t2,p2), Taskmap(t3,p3)
9.   }.
10. }
```

Figure 11. Design space generated by model finding procedure.

metamodels, feature diagrams, and platforms to domains, but study how domains can be used once this translation is accomplished.

3.3 Finite Model Finding

Throughout this paper we will show that domains can be formally analyzed for many purposes. FORMULA provides a powerful technique for supporting formal analysis called *finite model finding*. Conceptually, the procedure is simple:

Input:	A domain D and query q from D.
Output:	A finite syntactic instance X, such that q is satisfied if $R_0 = X$. Or, report that no such X exists.

(A finite syntactic instance is a finite set of terms from $\mathcal{T}_{\Upsilon_P}(\Sigma)$.)

Finite model finding for first-order logic is undecidable, so in some cases the procedure will neither be able to report success nor failure. FORMULA implements a unique model finding procedure [24] that combines *abduction* techniques from logic programming [17] with state-of-the-art *SMT* (SAT Modulo theories) solving [16]. The details of this procedure are beyond the scope of this paper.

The Taskmap abstraction illustrates the power of model fininding. Figure 9 shows two additional queries added to the Taskmap domain. The resc query is satisfied if there exists some resource constraints between tasks x, y and y, z. The threePath query is satisfied if the resc and conforms queries are both satisfied. The model finding procedure for the threePath query must construct syntactic instances containing at least three tasks that are well-scheduled. This amounts to solving a range of coloring problems, where the number of colors is the number of processors in the instance. Figure 10 shows one instance returned by FORMULA. It contains two colors (processors) and schedules the tasks accordingly.

Our model finding procedure not only constructs satisfying instances, but also determines a range of finite solutions by producing bounds on the number and types of terms that can appear in a satisfying instance. This range can be viewed as a *design space* of solutions to the query. Figure 11 shows the design space represented as an automatically generated domain called ThreeSpace. Line 2 declares the ThreeSpace domain as a *restriction* of the Taskmap domain, meaning its conforming instances also conform to the Taskmap domain. (This concept is explained fully in the next section.) The bounds block in Line 3 contains a finite set B of primitive terms (with variables). An instance X conforms to the ThreeSpace domain if X conforms to Taskmap and:

$$\exists \phi,\ \phi(B) = (R_\infty \cap \mathcal{T}_{\Upsilon_P}(\Sigma)). \tag{15}$$

In other words, if the logic program is executed for $R_0 = X$ then R_∞ contains all terms calculated by the logic program. Throw out all the terms in R_∞ that are not primitive terms, then there must be a substitution ϕ where $\phi(B)$ is exactly the remaining terms. The design space above captures all legal assignments of three tasks with constraints to anywhere from one to three processors. In summary, the model finding procedure also produces design spaces with relevant bounds on contents of satisfying models. Also, note that the bounds construct does not increase the expressiveness of FORMULA. It can be reduced to a finite set of LP expressions.

4 Composition of Syntaxes

Composing syntaxes is the process of building larger syntaxes from smaller ones. This process is important in model-based development, because rich syntaxes correspond to subproblems of the overall design problem. For example, the Taskmap abstraction represents the problem of scheduling tasks onto processors. Similarly, the ECU/Bus language (Figure 2) represents the problem of architecting a hardware substrate. Both of these problems must be solved to construct a complete system, and this requires joining together the two rich syntaxes in a meaningful way.

There already exists mechanisms for composing syntaxes, which are well-known to the model-based community. Many composition mechanisms take their inspiration from modern programming languages, borrowing concepts like *namespaces* and *interfaces* to tie together pieces of syntax. The *package merge* mechanism of UML 2.0 supports many styles of composition, including these. Our goal is neither to define the best mechanism for syntax composition, nor to provide a laundry-list formalizing all existing mechanisms. Instead, we provide a basic set of composition operators that can also be formally analyzed. Table 2 lists the basic composition operators available in FORMULA.

Includes. The includes operator is used to import the declarations of one domain into another domain:

domain D' includes D { ... }.

The resulting domain D' has

$$\Upsilon'_P \supseteq \Upsilon_P,\ \Upsilon'_R \supseteq \Upsilon_R,\ E' \supseteq E[conforms/D.conforms] \tag{16}$$

The notation $E[x_1/x'_1, \ldots, x_n/x'_n]$ denotes the expressions formed by replacing every occurrence of x_i in E with x'_i. Thus, domain D' has direct access to the declarations in D, but does not necessarily utilize the conformance rules of D because it is renamed to $D.conforms$. The includes operation is defined if the signatures of D' do not contain contradictory function symbol definitions, and the expressions E are non-recursive and stratified.

There are several variants of includes that make stronger statements about D'. The restricts keyword requires that no new primitives are introduces in D' and $D.conforms$ is implicitly conjuncted onto the *conforms* of D'. Let the $models(D_i)$ be the set of all finite syntactic instances that satisfy the conforms query of domain D_i. Then restricts enforces that:

$$models(D') \subseteq models(D). \tag{17}$$

The extends variant implicitly disjuncts $D.conforms$ onto the *conforms* of D', therefore:

$$models(D') \supseteq models(D). \tag{18}$$

Renaming. The *renaming operator* "as" gives new names to the function symbols and queries in a domain. The expression:

(D as X)

produces a domain D' with the same signatures and expressions as D, except that every occurrence of a function symbol and query name is prepended by "X.".

Pseudo-product. The *pseudo-product operator* "$*$" is a precursor for building the *categorical product* of domains. The expression:

(D$_1$ * D$_2$)

defines a domain D' where:

$$\begin{aligned}&\Upsilon'_P = \Upsilon^1_P \cup \Upsilon^2_P,\ \Upsilon'_R = \Upsilon^1_R \cup \Upsilon^2_R,\\ &E' = E^1[conforms/D_1.conforms] \cup \\ &\quad E^2[conforms/D_2.conforms] \cup \\ &\quad \{conforms\ \ {:}?\ D_1.conforms\ \&\ D_2.conforms.\}\end{aligned} \tag{19}$$

The pseudo-product has the property that if D_1 and D_2 have disjoint signatures and query names, then:

$$models(D') \cong models(D_1) \times models(D_2). \tag{20}$$

This is called the *categorical product*; it means that every model $X \in models(D')$ can be uniquely partitioned into two subsets X_1 and X_2 so that $X_i \in models(D_i)$. This construct is important, because it combines two domains into a larger one while guaranteeing no non-trivial interactions.

Pseudo-coproduct. The *pseudo-coproduct operator* "$+$" is a precursor for building the *categorical coproduct* of two domains. The expression:

(D$_1$ + D$_2$)

defines a domain D' where:

$$\begin{aligned}&\Upsilon'_P = \Upsilon^1_P \cup \Upsilon^2_P,\ \Upsilon'_R = \Upsilon^1_R \cup \Upsilon^2_R,\\ &E' = E^1[conforms/D_1.conforms] \cup \\ &\quad E^2[conforms/D_2.conforms] \cup \\ &\quad \{conforms\ \ {:}?\ D_1.conforms\ \text{XOR}\ D_2.conforms.\}\end{aligned} \tag{21}$$

Let the $models(D_i)$ be the set of all finite syntactic instances that satisfy the conforms query of domain D_i. The pseudo-product has the property that if D_1 and D_2 have disjoint signatures and query names, then:

$$models(D') \cong models(D_1) \uplus models(D_2). \tag{22}$$

This is called the *categorical coproduct*; it means that every model $X \in models(D')$ is either in $models(D_1)$ or $models(D_2)$, but never both. Again, this construct is important, because it combines two domains into a larger one while guaranteeing no non-trivial interactions.

Operator	Usage	Description
includes, restricts, extends operators	D' includes D, D' restricts D, D' extends D,	Imports the declarations of D into D' and while renaming the conforms query of D' to D'.conforms.
renaming operator "as"	D as X	Produces a new domain from D by replacing every occurrence of a function symbol $f(\ldots)$ with $X.f(\ldots)$ and every query name q with $X.q$.
pseudo-product operator "$*$"	$D_1 * D_2$	Produces a new domain D' by combining the specifications of D_1 and D_2, and then adding the query (conforms :? conforms$_{D_1}$ & conforms$_{D_2}$).
pseudo-coproduct operator "$+$"	$D_1 + D_2$	Produces a new domain D' by combining the specifications of D_1 and D_2, and then adding the query (conforms :? conforms$_{D_1}$ XOR conforms$_{D_2}$).

Table 2. Basic set of composition operators.

4.1 Properties of Compositions

Regardless of the approach, complex modeling processes are often plagued by the non-local effects of composition. In our framework compositions may yield unexpected results due to interactions between declarations and logic programs. A minimum requirement to ensure that composition did not introduce inconsistencies it to check non-emptiness of $models(D)$. The model finding procedure of FORMULA is suited for this task: Perform modeling finding on the conforms query to check non-emptiness.

However, checking non-emptiness of models is only one of the tools available in FORMULA. Many of the composition operators guarantee relationships between domains. For example, recall the use of restricts in precisely defining the design space of Figure 11. The composition operators can also be combined to guarantee relationships by construction. For example, given a family of domains $(D_i)_{i \in I}$ and an one-to-one renaming function $r : I \mapsto \Sigma$, then the categorical product can always be built by the construction:

$$\Big(D_1 \text{ as } r(1) * D_2 \text{ as } r(2) * \ldots * D_n \text{ as } r(n)\Big) \qquad (23)$$

where renaming is used to ensure disjointness of declarations. The categorical coproduct can be formed by a similar construction. In the next section we use the composition operators to formally compose more aspects of our automotive example.

4.2 Building the CarTask Domain

We put these techniques into practice to support platform-based design in the automotive domain. The first step is to specify the CarWare DSL (Figure 2). Figure 12 shows the primitive signatures for the CarWare DSL. The Car, ECU, and Msg concepts are encoded as $(n+1)$-ary function symbols, where n is the number of attributes (fields) per class in the metamodel. An extra argument is

```
/// Car hardware abstraction
domain CarWare {
  Car  : (
    name  : Id,
    model : String,
    make  : String
  ).
  Ecu  : (
    name         : Id,
    isCritical   : Bool,
    memorySize   : PosInteger
  ).
  Msg  : (
    name         : Id,
    maxSize      : PosInteger,
    ecuSends     : Bool,
    ecuReceives  : Bool
  ).
  Flex   : (Ecu,Ecu). Can   : (Ecu,Ecu).
  Contains  :  (parent  : Any,child  : Any).
...
```

Figure 12. Symbols for CarWare domain.

added to each function symbol to store an identifier. The relational concepts CAN and FlexRay are encoded as binary function symbols over terms. The bus class is abstract in the metamodel, so it becomes a derived symbol that is not explicitly declared. Finally, containment is captured by the binary symbol $Contains(\cdot,\cdot)$. A term $Contains(x,y)$ indicates that x contains y.

The CarWare domain is defined by the expressions shown in Figure 13. Lines 25-27 derive the legal forms of containment through the derived cancon (cancontain) symbol. Lines 22-23 relate the concrete Can and Flex terms to the abstract bus terms. Lines 29-35 give the domain constraints for this domain. Line 29 disallows Msgss that are neither sent nor received. Lines 30-31 require at least one Ecu per Car. Lines 32-33 require checks that all contain-

```
...
    /// Subtype relationships
22. bus(x) :- x is Can.
23. bus(x) :- x is Flex.
    /// Containment relationships
25. cancon(x,y) :- x is Car, y is Ecu.
26. cancon(x,y) :- x is Car, bus(y).
27. cancon(x,y) :- x is Ecu, y is Msg.
    /// Domain constraints
29. badMsg :? Msg(id,sz,false,false).
30. noECU :? x is Car,
31.          fail Contains(x, Ecu(_,_,_)).
32. badCon :? Contains(x,y),
33.           fail cancon(x,y).
34. conforms :? !noEcu & !badCon &
35.              !badMsg.
36. }
```

Figure 13. Domain constraints in Carware.

```
   /// Composition of abstractions
2. domain CarTask restricts
3.                ( Taskmap * CarWare) {
4.   noPrc :? Ecu(id,_,_),
5.            fail Processor(id).
6.   noEcu :? Processor(id),
7.            fail Ecu(id,_,_).
8.   conforms :? !noPrc & !noEcu.
9. }
```

Figure 14. Composition of Taskmap and CarWare.

ments are proper. Lines 34-35 aggregate all of the rules into the conforms query.

The CarTask domain combines the Taskmap and CarWare domains. Both pseudo-product and restriction operations are used to carefully build the composition, as shown in Figure 14. In Line 3 the product of the two domains is constructed. The product composition is an ideal starting point, because no interactions can occur between the domains. Next, the CarTask domain adds additional queries that restrict the product to models where Processors and Ecus are related by their unique IDs. Lines 4-5 search for Ecus without corresponding Processors; Lines 6-7 search for Processors without corresponding Ecus. Therefore, these additional domain constraints bind the two abstractions together by "equating" processors and ECUs.

The CarTask domain can be examined for consistency using model finding. First, we prove that the composition does not contain contradictions. Modeling finding on conforms yields the following well-formed model in the composition:

$$\left\{ \begin{array}{c} c = CarWare.Car(0, \text{“}a\text{”}, \text{“}b\text{”}), \\ e = CarWare.Ecu(2, \mathtt{false}, 3), \\ CarWare.Contains(c, e), \\ TaskMap.Processor(2) \end{array} \right\}$$

In conclusion, disciplined composition operators combined with model finding provides a framework to rigorously reuse rich syntaxes occurring in model-based development.

5 Transformations and Validation

Rich syntaxes can also be related through translators; a translator $\tau : models(D) \rightarrow models(D')$ takes a syntactic instance from domain D and translates it into a syntactic instance in domain D'. In traditional programming languages a translator may be a compiler that generates machine code from C++. In model-based development translators, also called *model transformations*, are used to: (1) Attach behavioral semantics to rich syntaxes [11] (i.e. semantic anchoring), (2) weave collections of specifications into a single whole [21] (i.e. aspect/model weaving), (3) and migrate models between platforms/abstraction layers. Transformations can be constructed from parts of other transformations, providing a natural mechanism for reuse. For example, the transformation $\tau(X)$:

$$\tau(X) = \tau_3(\tau_2(\tau_1(X)))$$

is τ_3 after τ_2 after τ_1.

5.1 Transformations

In our framework transformations are also described using non-recursive and stratified logic programs. A transformation τ is:

$$\tau = \langle D, D', \Upsilon_H, E_\tau \rangle \tag{24}$$

1. D is the input domain, and D' is the output domain; domains must have disjoint function symbols.
2. Υ_H is a finite signature contains *helper* function symbols (disjoint from those in D, D') used by the transformation.
3. E_τ is a LP that examines input terms from D and produces output terms in D'. It must that $E \cup E_\tau \cup E'$ is non-recursive and stratified.

A transformation is executed over a syntactic instance X from domain D. First, set $R_0 = X$ and execute the logic program $E \cup E_\tau \cup E'$. Next, drop all terms from R_∞ not in the output domain; this is the syntactic instance X'.

$$X' = R_\infty \cap \mathcal{T}_{\Upsilon'_P}(\Sigma) \tag{25}$$

For example, let $D = CareWare$, $D' = Taskmap$, $\Upsilon_H = \emptyset$, and $E_\tau = \{Processor(x) \leftarrow Ecu(x, y, z)\}$. This transformation takes a network of ECUs from the CarWare syntax and produces a set of processors in the Taskmap syntax with the same identifiers.

Figure 15 shows the notation for writing this transformation. Line 1 declares that the transformation Simple has input domain

```
transform Simple (CarWare) returns
                  (Taskmap){
  Processor(id) :- Ecu(id,_,_).
}
```

Figure 15. Simple transformation.

CarWare and output domain Taskmap. In this example no helper symbols are needed. Line 3 contains the single expression that produces a $Processor$ term for each Ecu term. Note that renaming operators can be used when the input and output domains are not sufficiently disjoint.

5.2 Structure Preserving Transformations

Transformations, like any other modeling artifact, may contain mistakes. An incorrect transformation may manifest itself by converting some well-formed input (w.r.t. D) to a malformed output (w.r.t. D'). In this case the transformation does not respect the rules of the rich syntax, which is a serious error. However, this property is not easily established if the input/output domains are rich syntaxes. For example, any transformation into the TaskMap domain must always yield a well-colored set of tasks. In the Simple transformation this is true because every output contains zero tasks, which is trivially well-colored. However, for more complex transformations this fact is not obvious.

Formally, a transformation τ characterizes a map:

$$[\![\ ]\!]^{\tau} : \mathcal{P}(\mathcal{T}_{\Upsilon_P}) \rightarrow \mathcal{P}(\mathcal{T}_{\Upsilon'_P}) \tag{26}$$

between the powersets of the term algebras of the input and output domains. We say that a transformation τ is *structure preserving* (SP) if:

$$\forall X \in models(D), \quad [\![X]\!]^{\tau} \in models(D'). \tag{27}$$

In other words, every well-formed instance in D is mapped to a well-formed instance in D'. Checking τ for the SP property is a form of *validation*; this guarantees that some properties are correct. However, there may be other errors not manifested at the syntactic level, which require additional techniques to discover [60, 28].

Our model finding procedure can determine if a transformation is not structure preserving. Transformations that are not SP will have some well-formed input that is malformed under the transformation. It is possible to search for such an input by constructing an *analysis domain* D_τ:

```
domain $D_\tau$ includes (D * $abstract$(D')) {
  $\Upsilon_H$, $E_\tau$,
  notSP :? D.conforms & !D'.conforms.
}
```

D_τ is formed by constructing a modified product of the input/output domains, and then combining this with the helper symbols Υ_H and transformation rules E_τ. The operation $abstract(\cdot)$ creates a domain where all of the primitive symbols have been changed to derived symbols:

$$abstract(D) = (\emptyset, \Upsilon_P \cup \Upsilon_R, E). \tag{28}$$

The query notSP is satisfied if there exists a syntactic instance composed entirely of terms from the input domain, such that D.conforms is satisfied and D'.conforms is not satisfied. If notSP fails, then the transformation is structure preserving. This property is due to the fact that all of the primitive symbols of D' have been changed to derived symbols, which forces the model finding procedure to consider only syntactic instances of the input domain. Furthermore, the only relationship between the input and output domains is through the transformation, therefore the model finding procedure must reason through the transformation rules. In the next section we show an application of to feature diagrams.

5.3 Example: Features

Features identify slices of a system that are not isolated from each other. Though feature diagrams capture the legal combinations of features, the engineer must relate each feature to a part of the implementation. In this section we show that features can be defined as transformations from a feature language to an implementation syntax. Transformations that are not structure preserving correspond to badly specified features.

Figures 16, 17 shows several features corresponding to slices of an automotive embedded system. This example is adapted from specifications developed by the *AUTOSAR* initiative, which is a consortium whose goal is a standard automotive embedded systems platform [3]. Briefly, The Car feature requires a DashboardECU which hosts the Alerter task. This task controls indicators on the dashboard that alert the driver to problems. The Maneuvering feature requires an EngineECU that hosts the EngineDynamics task. This task monitors the state of the engine, e.g. locations of pistons and temperature. The TorqueCalculator task, which must be located on a different ECU, consumes the engine data to calculate the torque generated by the engine. Finally, the CruiseControl feature needs a CruiseControl task residing on the CruiseECU. Additionally, a LeftSensor and RightSensor must be available to report exact conditions of the left/right wheels. The PowerTrainCoordinator task takes input from the Cruise Control and affects the state of the engine.

In this example features are both partial specifications and span implementation. For example, the Maneuvering feature introduces a TorqueCalculator, but does not specify where it must reside. Notice how the Automatic feature, which is at the bottom of the feature hierarchy, affects the TorqueCalculator task that was introduced at the top of the feature hierarchy. This is typical of the “cross-cutting” nature of features.

5.3.1 Defining the CarFeat Language

We begin by encoding the feature diagram as a domain, called CarFeat. Figure 18 shows the encoding, which reuses the intuition that a feature diagram is a grammar. The primitive symbol $Termin(\cdot)$ names the terminal features and the derived symbol $nonter(\cdot)$ names nonterminal features. Finally, the derived symbol $badimplies(\cdot,\cdot)$ is derived whenever feature x implies feature

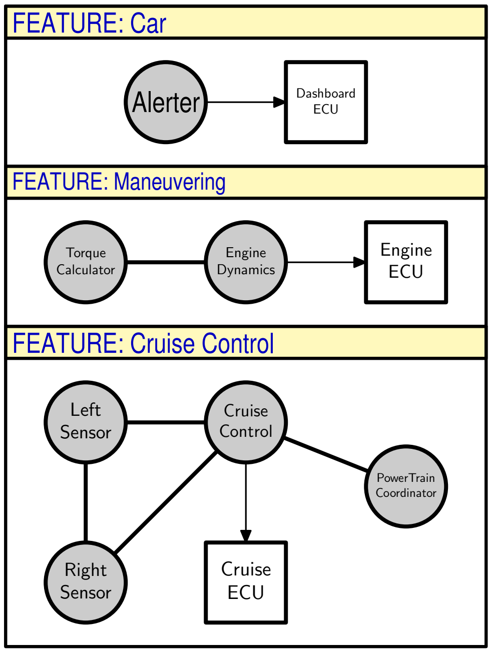


Figure 16. Relationships between features and implementation.

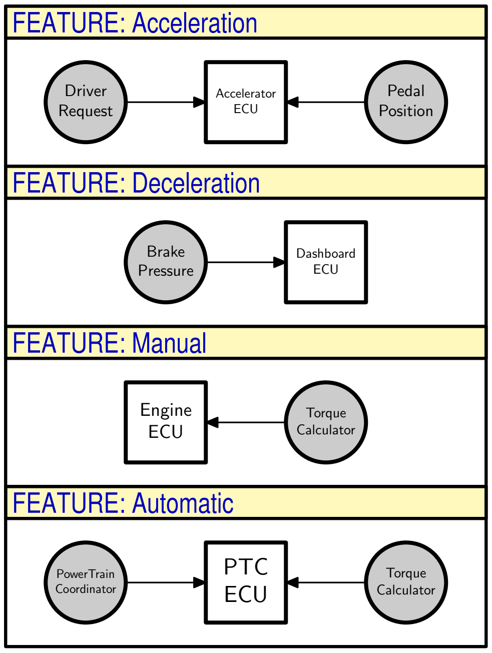


Figure 17. More features and implementation.

y, but y is not in the feature set. Lines 2-3 introduce a new finite type called `Feats` that enumerates the feature names. In the interest of space, the Deceleration feature is treated as terminal.

The expressions of the domain closely resemble a set of grammar productions. The Automatic and Manual features are mutually exclusive; two expressions encode this property (Lines 10-13). Implications between features are easy to capture with LP. Lines 14-15 create a *badimplies* term when the Cruise feature is used without the Autom feature. The nonterminal feature Car is produced whenever the Maneuvering feature is produced with no occurrences of *badimplies*. A set of features conforms to the CarFeat domain if the nonterminal $nontermin$(“Car”) is produced (Line 16).

5.3.2 The FeatMap Transformation

In this section we represent the relationship between features and partial specifications using FORMULA transformations. For instance, the expressions below:

$$\begin{aligned} Task(\text{“Alerter”}) &\leftarrow nonter(\text{“Car”}).\\ Processor(\text{“DashboardECU”}) &\leftarrow nonter(\text{“Car”}).\\ Taskmap(Task(\text{“Alerter”}),&\\ Processor(\text{“DashboardECU”})) &\leftarrow nonter(\text{“Car”}). \end{aligned}$$

generate the components associated with the Car feature. As a shorthand, FORMULA allows expressions with the same body to be combined into a single expression:

$$\begin{aligned} Task(\text{“Alerter”}),\ Processor(\text{“DashboardECU”}), \ldots\\ \leftarrow nonter(\text{“Car”}). \end{aligned}$$

Figure 19 shows a partial specification on the FeatMap transformation. Lines 4-11 define the nonterminal Maneuvering feature and Lines 13-20 define the terminal Automatic feature. The remaining features are omitted in the interest of space.

Though the FeatMap transformation is easily specified, its correctness depends on the input/output domains. For example, the Cruise feature introduces the PowerTrainCoordinator task without assigning it to a processor, which could indicate a mistake in the specification. However, the Automatic feature always accompanies Cruise, and Automatic does assign PowerTrainCoordinator to the PTC_ECU. Thus, correctness cannot be determined on a per feature basis, but must take into account the constraints on the feature language and the implementation syntax. This is precisely the structure preserving property described earlier.

Our example intentionally contained several mistakes. FORMULA detects these mistakes by constructing the analysis domain $D_{FeatMap}$ and then performing model finding over the query notSP. The procedure returns several interesting syntactic instances:

$$X_1 = \{termin(\text{“Manual”}), termin(\text{“Decel”})\} \tag{29}$$

and

$$X_2 = \left\{ \begin{array}{c} termin(\text{“Autom”}),\\ termin(\text{“Cruise”}),\\ termin(\text{“Decel”}) \end{array} \right\} \tag{30}$$

It also easy to find out why these instances are not structure preserving. Applying the $FeatMap$ transformation to X_1 yields the set of terms $X_1' = [\![X_1]\!]^{FeatMap}$. Next, we ask FORMULA to report the terms that satisfy the notConforms query:

notConforms :? !conforms

```
domain CarFeat {
  Feats : {"Car", "Maneuv", "Cruise",
   "Accel", "Decel", "Autom", "Manual"}.
  Termin : ( Feats ).
  /// Grammar derivations
  nonter("Car") :- nonter("Maneuv"),
    fail badimplies(_,_).
  nonter("Maneuv") :- nonter("Accel"),
    termin("Decel").
  nonter("Accel") :- termin("Autom"),
    fail termin("Manual").
  nonter("Accel") :- termin("Manual"),
    fail termin("Autom").
  badimplies("Cruise","Manual") :-
    termin("Cruise"), termin("Manual").
  conforms :? nonter("Car").
}
```

Figure 18. An automotive feature language.

```
transform FeatMap (CarFeat) returns
                (Taskmap){
  /// Maneuver mapping
  Task("TorqueCalc"),
  Task("EngineDyn"),
  Processor("EngineECU"),
  Constraint(Task("TorqueCalc"),
       Task("EngineDyn")),
  Taskmap(Task("EngineDyn"),
       Processor("EngineECU"))
                   :-nonter("Maneuv").
  /// Automatic control mapping
  Task("PowerTrainCoor"),
  Task("TorqueCalc"),
  Processor("PTC_ECU"),
  Taskmap(Task("PowerTrainCoor"),
       Processor("PTC_ECU")),
  Taskmap(Task("TorqueCalc"),
       Processor("PTC_ECU"))
                   :-nonter("Autom").
...
```

Figure 19. Transformational specification of features.

This yields the following set of terms and query assignments:

$$\left\{ \begin{array}{c} conforms = false, badBap = true, \\ Task(\text{“TorqueCalc”}), Task(\text{“EngineDyn”}) \end{array} \right\} \tag{31}$$

indicating that the TorqueCalculator and EngineDynamics tasks violated a scheduling constraint. Similarly, evaluating notConforms on $X_2' = [\![X_2]\!]^{FeatMap}$ produces the following trace:

$$\begin{array}{c} \{conforms = false,\ noMap = true,\ Task(\text{“LeftSensor”})\} \\ \{conforms = false,\ noMap = true,\ Task(\text{“RightSensor”})\} \end{array} \tag{32}$$

In this case, the LeftSensor and RightSensor tasks were not assigned to processors.

6 Design-Space Exploration

The abstractions and analysis techniques of model-based development are useful for automatically investigating many possible system architectures. This process is called *design-space exploration* (DSE). In order to apply DSE the user must characterize: (1) a set of architectures, called the *design space*, and (2) a fitness function that measures the optimality of a point design. DSE prunes the design space, using the fitness function, to present a near-optimal set of designs. The optimization of complex fitness functions is not unique to model-based design and spans many fields including nonlinear control theory [58], game theory [42], and artificial intelligence [62]. However, the success of DSE depends not only on the optimization techniques, but also on the representation of the design space. It is easy for the size of the space to become so large that it cannot be effectively explored. Existing techniques reduce the size of the space by approximating the set of *all interesting designs* with a set of *syntactically related designs*, thereby making exploration feasible [43, 15].

We use the model finding procedure implemented in FORMULA [24] to compactly represent design spaces defined over a rich syntax. This procedure converts a goal/domain pair (G, D) to a Boolean formula $\phi_G(\overline{x})$, where $\overline{x}$ is a set of Boolean variables. Each solution s to ϕ_G corresponds to an interesting instance solving the goal; ϕ_G may have an exponential number of solutions (in the size of $\overline{x}$). Thus, the Boolean formulas should be viewed as a compact representation of a space of solutions. The n^{th} point design can be lazily constructed by conjuncting the (Boolean) negation of the previous $n-1$ solutions $s_1, \ldots, s_{n-1}$ to ϕ_G and solving ϕ_n:

$$\phi_1 = \phi_G, \quad \phi_{n>1} = \phi_G \wedge \bigwedge_{1 \leq i \leq n-1} (\neg s_i) \tag{33}$$

Or, the formulas can be converted to *Boolean decision diagrams* (BDDs) permitting easy enumeration of solutions. Both of these approaches have been used in design space exploration.

6.1 A Design Space Representation

In this section we explore possible architectures for a network of ECUs connected by Buses so that the overall system is (1) fault tolerant, (2) minimizes cabling (physical length of wires), and (3) maximizes throughput. We focus on constructing the design space over which exploration takes place. Techniques for formulating and evaluating a reasonable fitness measure can be found in [15].

The first step towards DSE is a suitable rich syntax for expressing relevant architectures. We have already developed such a rich syntax, called the CarTask domain (Figure 14), which represents networks of ECUs combined with tasks. Given a set of tasks T and processors P, the design space should be a set of CarTask instances, each of which contains T and P along with some bus architecture. However, the original CarTask domain did not enforce that ECUs should be connected; so additional expressions

```
   /// Car exploration domain
2. domain CarExplore restricts CarTask {
3.   hop1(x,z), hop1(z,x) :-
4.          bus(x,z), x != z.
5.   hop2(x,y,z) :-
6.          hop1(x,y), hop1(y,z), x != z.
7.   hop3(w,x,y,z) :- hop2(w,x,y),
8.          hop1(y,z), z != w, z != x.
9.   tooFar :? x is Processor,
10.   y is Processor, z is Processor,
11.   w is Processor, fail hop1(w,x),
12.   fail hop2(w,x,y), fail hop3(w,x,y,z).
13.  tooClose :? hop1(x,y), hop1(y,z),
14.               hop1(z,x).
15.  conforms :? !tooFar & !tooClose.
16. }
```

Figure 20. Restricting the CarTask **domain for DSE.**

```
hasFeats :? FeatMap({Termin("Cruise"),
   Termin("Autom"), Termin("Decel")}).
explore :? hasFeats & conforms.
```

Figure 21. Constructing a design space

must be added to the CarTask domain to define a reasonable design space.

Figure 20 shows the additional expressions needed for DSE over bus topologies. The derived symbols $hop1$, $hop2$, $hop3$ record the distance between any two processors in the network. Two processors x and z that are farther than three hops will not derive any of these terms. Lines 3-8 define hops using *disequality* constraints, which require two variables to take distinct values. Topologies that are too weakly connected or too strongly connected (too much cabling) are characterized with the tooClose and tooFar queries. Two processors x and z are too far away if they a farther than three hops (Lines 9-12). Three processors x, y, and z are too close if they form a triangle (Lines 13-14). Line 2 declares that the CarExplore domain is a proper restriction of the CarTask domain; conformance is modified with these additional topology requirements (Line 15).

Constructing the design space is now a matter of writing a query in the CarExplore domain. Figure 21 shows a query explore that constructs a design space for a Car containing the Cruise control and Automatic transmission features. The query hasFeats is created by applying the the FeatMap transformation to the input terms $\{Termin(\text{“Autom”}), Termin(\text{“Cruise”}), Termin(\text{“Decel”})\}$. The output of this transformation is the body of the query. The explore query requires implementations of the feature set with bus topologies that are neither too dense nor too sparse.

Ground Term	Variable
$Processor(\text{“EngineECU”})$	a_1
$Processor(\text{“DashboardECU”})$	a_2
$Processor(\text{“CruiseECU”})$	a_3
$Processor(\text{“PTCECU”})$	a_4
$Task(\text{“LeftSensor”})$	a_5
$Task(\text{“RightSensor”})$	a_6
$Constraint(\ldots(\text{“LeftSensor”}), \ldots(\text{“RightSensor”}))$	a_7

Table 3. Ground terms appearing in goal.

6.2 Details of the Design Space

We now summarize the resulting design space constructed by model finding. The procedure attempts to calculate a finite set of candidate terms S with the property that if there exists any solution to the query, then there must also exist a solution $S' \subseteq S$. Once the set S is calculated, each term $t_i \in S$ is converted to a Boolean variable b_i, and a Boolean formula ϕ_G is generated over these variables. If b_i is true in satisfying assignment of ϕ_G, then the term t_i is in the solution; otherwise t_i is not in the solution.

Intuitively, the set of candidate terms is built in several phases. First, non-negated terms without variables, *called ground terms*, appearing in the query must be part of any solution; these terms are immediately included in S. Table 3 shows some of the ground terms introduced by the FeatMap transformation. The *Variable* column lists the Boolean variable associated with each term. Next, the procedure finds ground terms that must be considered because of negations. For example, the query $\neg toofar$ will be considered for $w = Processor(\text{“CruiseECU”})$ and $x = Processor(\text{“EngineECU”})$. This causes $hop1(w, x)$ to be considered, resulting in the conclusion that $bus(\ldots(\text{“CruiseECU”}), \ldots(\text{“EngineECU”}))$ may be in some solution. Table 4 shows ground terms introduced by this process. Notice that these terms include all the possible Buses that may appear in the architecture.

Model finding may also introduce terms that contain variables. Recall that the Cruise control feature contained tasks LeftSensor and RightSensor that were not assigned to any processor. However, these tasks must be assigned to processors, otherwise noMap will be satisfied. In response, FORMULA introduces new variables p_1 and p_2 that stand for these missing processors. Table 4 also shows some of the non-ground terms.

After the candidate set S has been constructed, a Boolean formula ϕ_G is produced that encodes the query. In the interest of space, we only sketch ϕ_G. The ground terms in Table 3 appear in the query and must be in any solution, leading to the simple encoding:

$$\phi_G^I = a_1 \wedge a_2 \wedge \ldots \wedge a_7 \tag{34}$$

The domain constraints require an Ecu term for each $Processor$ term, and a $Taskmap$ term for each $Task$. The encoding forces

Ground Term	Variable
$bus(\ldots(\text{“EngineECU”}), \ldots(\text{“DashboardECU”}))$	b_1
$bus(\ldots(\text{“CruiseECU”}), \ldots(\text{“EngineECU”}))$	b_2
$bus(\ldots(\text{“EngineECU”}), \texttt{PTCECU})$	b_3
$bus(\texttt{PTCECU}, \ldots(\text{“DashboardECU”}))$	b_4
$bus(\texttt{PTCECU}, \ldots(\text{“CruiseECU”}))$	b_5
$bus(\ldots(\text{“DashboardECU”}), \ldots(\text{“CruiseECU”}))$	b_6

Non-ground Term	Variable
$Ecu(\text{“EngineECU”}, x_1, y_1)$	c_1
$Ecu(\text{“CruiseECU”}, x_2, y_2)$	c_2
$Processor(p_1)$	c_3
$Processor(p_2)$	c_4
$Taskmap(\ldots(\text{“LeftSensor”}), Processor(p_1))$	c_5
$Taskmap(\ldots(\text{“RightSensor”}), Processor(p_2))$	c_6

Table 4. Terms implied by negations.

these terms to appear in pairs:

$$\phi_G^{II} = \begin{bmatrix} (a_1 \wedge c_1) \vee (\neg a_1 \wedge \neg c_1) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_2 \wedge c_2) \vee (\neg a_2 \wedge \neg c_2) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_5 \wedge c_5) \vee (\neg a_5 \wedge \neg c_5) \end{bmatrix} \wedge \\ \begin{bmatrix} (a_6 \wedge c_6) \vee (\neg a_6 \wedge \neg c_6) \end{bmatrix} \wedge \ldots \tag{35}$$

Care must be taken when mapping tasks LeftSensor and RightSensor due to scheduling conflicts:

$$\phi_G^{III} = (p_1 \neq p_2) \vee (\neg a_7 \vee \neg c_5 \vee \neg c_6) \tag{36}$$

This is not strictly a Boolean formula, because $p_1 \neq p_2$ is a disequality over non-Boolean variables. These non-Boolean disequalities can also be reduced to Boolean variables [10]; for simplicity we write the formula in this extended form. Next, buses may be too close:

$$\phi_G^{IV} = \begin{array}{l} (\neg b_1 \vee \neg b_2 \vee \neg b_6) \wedge (\neg b_1 \vee \neg b_3 \vee \neg b_4) \wedge \\ (\neg b_2 \vee \neg b_3 \vee \neg b_5) \wedge (\neg b_4 \vee \neg b_5 \vee \neg b_6) \wedge \ldots \end{array} \tag{37}$$

Finally, processors must be close enough to each other. This rule generates a large subformula; only a small portion is shown:

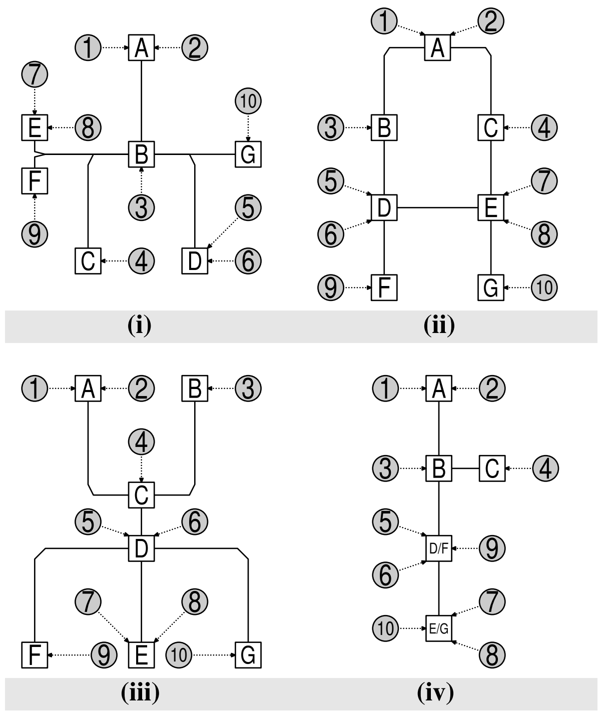
$$\phi_G^{V} = (a_1 \wedge a_2 \wedge b_1) \vee (a_1 \wedge a_3 \wedge a_2 \wedge b_2 \wedge b_6) \vee \ldots \tag{38}$$

The entire design space is represented by the conjunction of these formulas:

$$\phi_G = \phi_G^{I} \wedge \phi_G^{II} \wedge \phi_G^{III} \wedge \phi_G^{IV} \wedge \phi_G^{V} \tag{39}$$

Currently the SMT solver *Z3* [16] is used to evaluate these formulas. Z3 also chooses reasonable values for non-Boolean variables, like processor IDs, when it returns a satisfying assignment.

Figure 22 shows some of the interesting architectures in this design space. The first design is a *star* topology; all messages pass through the EngineECU. The topology of the second design splits the system into two subnetworks with a strategically placed bridge between the PTC and Accelerator ECUs. Without the bridge, some ECUs would be too far from each other. Design three takes a different approach, and splits the network into two *star* topologies with a bridge between the subnetworks. Finally, the fourth design is the most serialized design possible. This architecture is possible because the left and right sensors are scheduled onto existing ECUs, instead of instantiating new ones.



Key	Value
A	`DashboardECU`
1	`Alerter`
2	`BrakePress`
B	`EngineECU`
3	`EngineDyn`
C	`CruiseECU`
4	`Cruise`
D	`PTCECU`
5	`Coordinator`

Key	Value
6	`TorqueCalc`
E	`AccelECU`
7	`DriverReq`
8	`PedalPos`
F	p_1
9	`LeftSensor`
G	p_2
10	`RightSensor`

Figure 22. Members of the design space.

7 Related Work

Formalizations and applications of rich syntax have appeared in many different forms. Within the domain-specific language community, graph-theoretic formalisms [18, 7, 48] have received the most research attention. However, the majority of work focuses on graph rewriting systems as a foundation for model transformations. See [40, 32] for a taxonomy of existing graph-theoretic model transformation approaches. The problems of calculating properties of rich syntax, composing syntax with known properties, and constructing design space representations have not received the same attention from graph-theoretic methods. For example, the model transformation tool VIATRA [12] supports executable Horn logic (i.e. Prolog) to specify transformations, but does not focus on restricting expressiveness for the purpose of analysis.

The visibility of UML has driven researchers to formalize it semantics. This is a non-trivial task because UML includes many capabilities (diagrams) including metamodeling, state machines, activities, sequence charts (interactions), and use-case diagrams [47]. Approaches for formalizing UML must tackle the temporal nature of its various behavioral semantics, necessitating more expressive formal methods. Well-known tools/methods such as Alloy [22], B [37], and Z [19] have been used to varying degrees of success. These approaches make trade-offs between expressiveness and the degree of automated analysis. For example, Z and B proofs typically require interactive theorem provers [9, 4] and model finding may not be supported. Z or B formalizations of UML could be a vehicle for studying rich syntax, but automated analysis is less likely to be found.

Alloy, like FORMULA, is less expressive than other methods, thereby supporting automated analysis [23]; it also has a recently improved model finding procedure [59]. However, the mathematical underpinnings of Alloy are quite different from FORMULA: Alloy supports first-order logic with relations over atoms plus transitive closure. Contrarily, our framework is based on a non-monotonic extension of Horn logic [24]. One key difference is that FORMULA specifications can be executed like standard *logic programs* [53]. Complexity-theory also offers a coarse-grained way of comparing logic programs with other methods [14].

The BNF grammars of traditional programming languages can be extended to capture richer syntaxes. *Attribute grammars* (AGs) [49] , proposed by Knuth [31], could be the earliest example of such a mechanism. AGs allow the productions of a BNF grammar to trigger actions capable of examining tokens and attaching new data to tokens. These actions can be specified programmatically, thereby significantly increasing the power of the grammar. However, calculating properties of languages specified through AGs depends on the expressiveness of the actions. Additionally, composing AGs has proved to be a difficult task [20]. More recently, *pluggable type systems* have been studied as a mechanism to compose the type systems of traditional programming languages [2].

Tools for creating and editing models (*CRUD* tools) [33, 57] intersect with databases, because they require a persistence layer for storing many different models across many domains. Various extensions of Horn logic have been utilized by the *declarative database* community [41] as powerful query languages; this work fits naturally with our notion of a *domain*. Functional programming has also been used to declaratively operate on databases [35]. The *Language Integrated Query* (LINQ) project extends this work, allowing in-memory data structures to be queried just like databases [39].

Numerous examples of structural representations for design spaces can be found in the literature. For example, [27] performs DSE for arbitrary algorithms by extracting a data dependency graph from the steps of a given algorithm A. The design space is the set all similar dependency graphs not contradicting the dependencies of the original algorithm. Dependency graphs form a syntactic construct approximating an ideal design space (consisting of all algorithms A' that compute the same function as A). The model-based tool DESERT [43] compactly represents automotive design spaces using AND-OR trees to encode architectural choices. These trees are converted to BDDs allowing the design space to be pruned without explicit enumeration of its elements. Recent work on fitness functions for evaluating automotive design spaces can be found in [15].

8 Conclusion

In conclusion, we have shown that Horn logic extended with negation provides a powerful framework for integrating the rich syntaxes and transformations occurring in model-based development. By restricting our focus to a well-understood logic-based kernel, we obtain many important and efficient analysis techniques. Specifically, we provide sound mechanisms for composing syntaxes, determining properties of compositions, detecting mistakes in model transformations, and constructing design spaces over rich syntax. These techniques have been implemented in FORMULA.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[3] AUTOSAR. Achievements and exploitation of the autosar development partnership. Technical report, 2006.

[4] R. Banach and S. Fraser. Retrenchment and the b-toolkit. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, pages 203–221, april 2005.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[6] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

[7] J. Bezivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *In Proceedings of the 16th Conference on Automated Software Engineering*, pages 273–280, 2001.

[8] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. Technical Report TIK Report-Nr. 16, Gloriastrasse 35, 8092 Zurich, 1996.

[9] A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.

[10] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Log.*, 3(4):604–627, 2002.

[11] K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *DATE*, pages 906–911, 2007.

[12] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra - visual automated transformations for formal verification and validation of uml models. In *ASE*, pages 267–270, 2002.

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, pages 211–220, 2006.

[14] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[15] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *DAC*, pages 278–283, 2007.

[16] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *In Proceedings of Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, March 2008.

[17] M. Denecker and D. D. Schreye. Sldnfa: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.

[18] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inform.*, 74(1):31–61, 2006.

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

[20] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: support for modularity in translator design and implementation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–234, 1992.

[21] J. G. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD*, pages 36–45, 2004.

[22] D. Jackson. A comparison of object modelling notations: Alloy, uml and z. Technical report, August 1999.

[23] D. Jackson. Automating first-order relational logic. In *SIGSOFT FSE*, pages 130–139, 2000.

[24] E. K. Jackson and W. Schulte. Model generation for horn logic with stratified negation. In *Proceedings of the 28th International Conference on Formal Techniques for Networked and Distributed Systems*, 2008.

[25] E. K. Jackson and J. Sztipanovits. Towards a formal foundation for domain specific modeling languages. *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT'06)*, pages 53–62, October 2006.

[26] M. Jackson and P. Zave. Domain descriptions. In *Proceedings of the IEEE Conference on Requirements Engineering*, pages 56–64, 1993.

[27] S. Kakita, Y. Watanabe, D. Densmore, A. Davare, and A. L. Sangiovanni-Vincentelli. Functional model exploration for multimedia applications via algebraic operators. In *ACSD*, pages 229–238, 2006.

[28] G. Karsai and A. Narayanan. On the correctness of model transformations in the development of embedded systems. In *Monterey Workshop*, pages 1–18, 2006.

[29] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[30] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[31] D. E. Knuth. The genesis of attribute grammars. In *In Proceedings of Attribute Grammars and their Applications*, pages 1–12, 1990.

[32] A. Königs and A. Schürr. Multi-domain integration with mof and extended triple graph grammars. In *Language Engineering for Model-Driven Software Development*, 2004.

[33] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. *Workshop on Intelligent Signal Processing*, May 2001.

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

[35] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[36] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, pages 176–187, 2002.

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

[38] D. Marx. Graph coloring problems and their applications in scheduling. *Periodica Polytechnica Ser. El. Eng.*, 48(1-2):5–10, 2004.

[39] E. Meijer, B. Beckman, and G. M. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 706, 2006.

[40] T. Mens, P. V. Gorp, D. Varró, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electr. Notes Theor. Comput. Sci.*, 152:143–159, 2006.

[41] J. Minker. Logic and databases: A 20 year retrospective. In *Logic in Databases*, pages 3–57, 1996.

[42] R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, September 1997.

[43] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[44] Object Management Group. Meta object facility specification v1.4. Technical report, 2002.

[45] Object Management Group. Mda guide version 1.0.1. Technical report, 2003.

[46] Object Management Group. Ocl specification, v2.0. Technical report, 2006.

[47] Object Management Group. Omg unified modeling language (omg uml), superstructure, v2.1.2. Technical report, 2007.

[48] F. Orejas, H. Ehrig, and U. Prange. A logic of graph constraints. In *FASE*, pages 179–198, 2008.

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

[50] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Design Autom. Electr. Syst.*, 11(3):537–563, 2006.

[51] C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.

[52] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. *Future of Software Engineering, 2007. FOSE '07*, pages 55–71, May 2007.

[53] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–21, 1989.

[54] A. Sangiovanni-Vincentelli, L. Carloni, F. D. Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In *Proceedings of the Design Automation Conference (DAC'04)*, June 2004.

[55] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.

[56] F. Sethna, E. Stipidis, and F. Ali. What lessons can controller area networks learn from flexray. *Vehicle Power and Propulsion Conference, 2006. VPPC '06. IEEE*, pages 1–4, 6-8 Sept. 2006.

[57] The Eclipse Modeling Framework. www.eclipse.org/emf/. Technical report.

[58] C. Tomlin, J. Lygeros, and S. Sastry. Computing controllers for nonlinear hybrid systems. In *HSCC*, pages 238–255, 1999.

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

[60] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[61] S. Wang, S. K. Birla, and S. Neema. A modeling language for vehicle motion control behavioral specification. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, pages 53–60, 2006.

[62] L. D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information & Software Technology*, 43(14):817–831, 2001.