Constructive Techniques for Meta- and Model-level Reasoning

Ethan K. Jackson and Janos Sztipanovits[1]

Institute for Software Integrated Systems,
Vanderbilt University, Nashville, TN 37235, USA
ejackson@isis.vanderbilt.edu
janos.sztipanovits@vanderbilt.edu

Abstract. The structural semantics of UML-based metamodeling were recently explored[1], providing a characterization of the models adhering to a metamodel. In particular, metamodels can be converted to a set of constraints expressed in a decidable subset of first-order logic, an extended Horn logic. We augment the constructive techniques found in logic programming, which are also based on an extended Horn logic, to produce constructive techniques for reasoning about models and metamodels. These methods have a number of practical applications: At the meta-level, it can be decided if a (composite) metamodel characterizes a non-empty set of models, and a member can be automatically constructed. At the model-level, it can be decided if a submodel has an embeddeding in a well-formed model, and the larger model can be constructed. This amounts to automatic model construction from an incomplete model. We describe the concrete algorithms for constructively solving these problems, and provide concrete examples.

1 Preliminaries - Metamodels, Domains, and Logic

This paper describes constructive techniques, similar to those found in logic programming, for reasoning about *domain-specific modeling languages* (DSMLs) defined with metamodels. Before we proceed, we must describe how a metamodel can be viewed as a formal object that characterizes the well-formed models adhering to that metamodel. We will refer to the models that adhere to metamodel X as *the models of metamodel X*. In order to build some intuition for this view, consider the simple *DIGRAPH* metamodel of Figure 1. The models of *DIGRAPH* consist of instances of the *Vertex* and *Edge* classes such that *Edge*

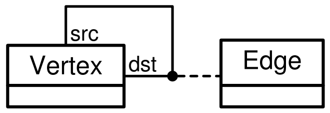

Fig. 1. DIGRAPH: A simple metamodel for labeled directed graphs

instances "connect" *Vertex* instances. In anther words, *DIGRAPH* characterizes a class of labeled directed graphs. Thus, a model might be formalized as a pair $G = \langle V \subseteq \Sigma, E \subseteq V \times V \rangle$, where Σ is an alphabet of vertex labels. If Σ is fixed, then the set $\mathscr{G}$ of all models of *DIGRAPH* is: $\mathscr{G} = \{(V, E) | V \subseteq \Sigma, E \subseteq V^2\}$. This is the classic description of labeled digraphs, and at first glance it might appear possible to extend this description to characterize the models of arbitrary metamodels. Unfortunately, UML-like metamodels[2][3] contain a number of constructs that deny a simple extension of graph-based descriptions. The *UNSAT* metamodel of Figure 2 illustrates some of these constructs. First, classes

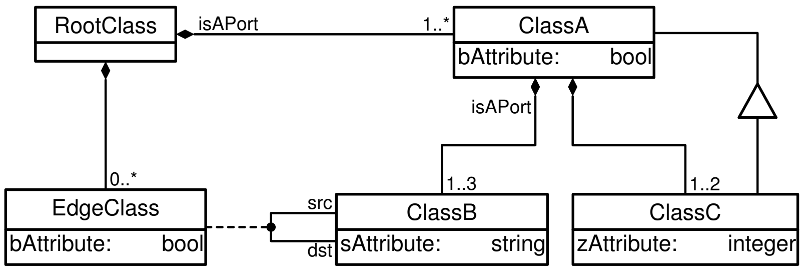

Fig. 2. UNSAT: A complex metamodel with no finite non-trivial models

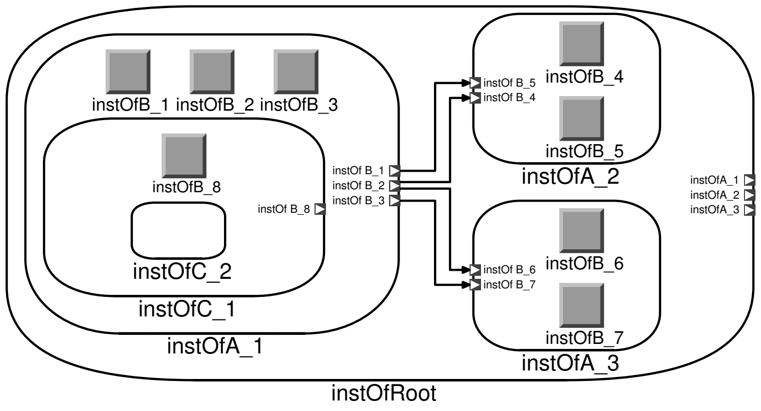

Fig. 3. Model that (partially) adheres to the UNSAT metamodel

may have non-trivial internal structure. For example, classes of *UNSAT* have typed member fields (called *attributes*). An instance of *ClassA* has a **boolean** field named *bAttribute*. Classes also inherit this structure, e.g. an instance of *ClassC* has two attributes, *bAttribute* and *zAttribute*, via inheritance. Instances may contain other instances with constraints on the type and number of contained instances. An instance of *ClassA* must contain between 1 and 3 instances of *ClassB*. Second, internal instance structure can be "projected" onto the outside of an instance as *ports*. The containment relation from *ClassA* to *RootClass* has the *isAPort* rolename, requiring that all contained instances of *ClassA* appear as interfaces on the outside of the containing instance of *RootClass*. Figure 3 shows a model with containment and ports. The hollow oblong shapes denote instances that can contain other instances, and the small squares with white arrows on the oblongs' borders denote ports. For example, the outermost container *instOfRoot* is an instance of the *RootClass* and contains three instances of *ClassA*. Each *ClassA* instance appears as a port on the far right-hand side of *instOfRoot*. Containment and ports are a useful form of information hiding, but they also complicate matters because ports permit edges to cross hierarchy. For example, the edges in Figure 3 connect instances of *ClassB* together even though these instances are not contained in the same instance. Furthermore, the edges are actually contained in the *RootClass* instance, even though the endpoints are not. The third major complication arises because edges are not simple binary relations. In UNSAT, edges are instances of *EdgeClass*, and so each edge has a member field named *bAttribute*. In general, edges must be distinguishable (i.e. labeled), otherwise it would not be possible to reliably determine the values of member fields. In fact, the UML-notation (correctly) implies that edges are ternary associations between an edge label, source label, and destination label.

Graph-based formalisms have been used extensively by the model transformation community, and provide reasonable approximations of model structure for the purpose of transformation. However, in this paper we do not focus on model transformation, but rather we explore techniques for reasoning about all the details of metamodel and model structure. One approach to characterizing realistic model structure might be to combine all existing graph extensions and consider models to be *hierarchical*[4], *typed, attributed*[5] *hypergraphs* with labeled edges. However, even this would not handle all aspects of modern metamodeling languages, and it would produce a brittle and unwieldy formalism. In [1] we present an alternative approach to model structure based on formal logic, which we briefly outline now. In order to present our view, we begin with the concept of a *domain* (in the sense of *domain-specific modeling languages*). A domain $D = \langle \Sigma, \Upsilon, \Upsilon_C, C \rangle$ is a quadruple where Σ is an (infinite) alphabet for distinguishing model elements, Υ is a finite signature for encoding model concepts, Υ_C is a finite signature for encoding model properties, and C is a set of logical statements (constraints) for deriving model properties. A *model realization* is set of terms from the *term algebra*[6] $T_{\Upsilon}(\Sigma)$ over signature Υ generated by Σ. The set of all possible model realizations is $\mathcal{P}(T_{\Upsilon}(\Sigma))$, i.e. all subsets of terms. We will use the notation $(f, n) \in \Upsilon$ to indicate that *function symbol* f of arity n is a member of the signature Υ.

Example 1. The domain of labeled digraphs $D_{\mathcal{G}}$ has the model realizations given by the signature $\Upsilon = \{(v, 1), (e, 2)\}$ and a countably infinite alphabet ($|\Sigma| = |\aleph_0|$). These two symbols to encode the concepts of vertex and edge. Vertices are encoded using the unary function symbol v and edges are encoded using the binary function symbol e. Some model realizations include:

1. $M_1 = \{\ v(\mathtt{c}_1), v(\mathtt{c}_2), e(\mathtt{c}_1, \mathtt{c}_2)\ \}$, a 2-path from a vertex $\mathtt{c}_1$ to a vertex $\mathtt{c}_2$.
2. $M_2 = \{\ v(\mathtt{c}_3), e(\mathtt{c}_3, \mathtt{c}_4)\ \}$, a dangling edge starting at vertex $\mathtt{c}_3$.
3. $M_3 = \{\ v(e(\mathtt{c}_5, \mathtt{c}_6)), v(v(\mathtt{c}_7))\}$, a structure that is not a graph at all.

where the symbols written in `typewriter` font indicate members of the alphabet.

The term algebra easily captures arbitrary n-ary concepts and permits concepts to be combined in complex ways. Example 1.3 shows that function symbols can be arbitrarily nested. This example also shows that not all model realizations combine the modeling concepts in ways that match our intentioned meaning of the symbols. Example 1.1 describes a simple 2-path, but 1.2 describes a dangling edge because vertex $\mathtt{c}_4$ is not in the model. Example 1.3 does not correspond to a graph in any obvious way, but is still a legal member of $\mathcal{P}(T_\Upsilon(\Sigma))$.

The set of model realizations of a domain contains all possible ways that the concepts can be used together. In fact, with a single operator f of arity greater than or equal to one, and an alphabet with at least one element, a countably infinite number of terms can be generated. (Consider a successor operation *succ* and $\Sigma = \{0\}$.) Thus, for all non-trivial cases the number of possible model realizations is uncountably infinite. Therefore $\mathcal{P}(T_\Upsilon(\Sigma))$ will typically contain many model realizations that use the function symbols contrarily to our intentions. In order to counteract this, we must define a set of model properties, characterized by another signature Υ_C, and set C of logical statements for deriving model properties. For simplicity, we assume that Υ_C simply extends the signature of Υ (i.e. $\Upsilon_C \supset \Upsilon$). For example, the property of directed paths could be captured by: $\Upsilon_C = \{(v,1),(e,2),(path,2)\}$ and $C = \{\forall x, y, z\ (e(x,y) \vee path(x,y)) \wedge e(y,z) \Rightarrow path(x,z)\}$. The symbol $path(\cdot,\cdot)$ encodes the concept of a directed path between two vertices. The single logical statement in C defines how to derive the paths in a digraph. They keyword *derive* is important, and there are some subtle points to be made about derivation.

Classically, the notion of a derivation is represented by a *consequence operator*, written $\vdash$, which maps sets of terms to sets of terms $\vdash$: $\mathcal{P}(T_{\Upsilon_C}(\Sigma)) \rightarrow \mathcal{P}(T_{\Upsilon_C}(\Sigma))$. A consequence operator encapsulates the inference rules of a particular style of logic, and may make use of additional *axioms* to derive terms. In our framework, the set C is the set of axioms that the consequence operator may use. Given a model M (i.e., a set of terms), $M \vdash_C M'$ denotes the set of terms M' that can be discovered from the terms M and the axioms C. A term t can be derived from a model M if $t \in M'$. We will simply write $M \vdash t$ to denote that $t \in M'$. Notice that the consequence operator generalization does not require terms to be viewed as predicates. For example, given the simple graph M_1 of Example 1.1, we can derive the term $path(\mathtt{c}_1, \mathtt{c}_2)$, but this term does not evaluate to a boolean value. Classical consequence operators, in the sense of Tarski, correspond to *closure operators* and are *extensive*, *isotone*, and *idempotent*[6]. Later, we will discuss the consequence operators of nonmonotonic logics where the isotone property does not hold. The history of mathematical logic is rich and diverse; we will not summarize it here. Instead, we will focus on particular applications and limit our discussion to those applications. For the reader unfamiliar with this area, it suffices to remember these two points: First, consequence operators capture the derivation of terms. Second, terms are not predicates.

Among the properties that can be encoded using Υ_C and C, we require at least one property to be defined that characterizes if a model is well-formed. We permit well-formedness to be defined either positively or negatively. A *positive domain* includes the function symbol $wellform(\cdot)$ in Υ_C, and a model M is well-formed if $\exists x \in T_{\Upsilon_C}(\Sigma),\ M \vdash_C wellform(x)$. In another words, a model is well-formed if a term of the form $wellform(x)$ can be derived for some x. A *negative domain* is characterized by the function symbol $malform(\cdot)$ such that a model is well-formed if $\forall x \in T_{\Upsilon_C}(\Sigma),\ M \nvdash_C malform(x)$. In another words, a model is well-formed if it is not possible to prove $malform(x)$ for any x. At first glance, it may appear that the positive domains have weaker definitions than negative domains. In fact, this depends on the expressiveness of the underlying logic of $\vdash$. For example, if the logic has a "negation" (which is not the usual propositional negation) then we can define $wellform(x) \Leftrightarrow \forall y\ \neg malform(y)$ for some arbitrary x. On the other hand, if the logic is restricted, then the positive domains may be strictly weaker than the negative domains.

A domain captures the set of possible model realizations and provides a mechanism to discern the good models from the bad ones. From this perspective, the set of all metamodels also defines a domain D_{meta} that characterizes all well-formed metamodels. Let the set $\mathcal{V}$ be a fixed vocabulary of function symbols and the sets Σ and Σ_v be two fixed disjoint countably infinite alphabets. Let $SIG(\mathcal{V}) = \{\Upsilon | \Upsilon : \mathcal{V} \to \mathbf{Z}_+\}$, be the set of all partial functions from $\mathcal{V}$ to the positive integers, i.e., the set of all possible signatures. Finally, let $\mathcal{F}(\Upsilon, \Upsilon_C)$ be the set of all formulas that can be defined over terms composed from function symbols of Υ, Υ_C with constants from Σ and variables from Σ_v. These parameters allows us to characterize the set of all domains $\Delta_{\mathcal{F}}$ that can be defined with a particular style of logic[1]:

$$\Delta_{\mathcal{F}} = \bigcup_{\Upsilon \in SIG(\mathcal{V})} \ \bigcup_{\Upsilon \subset \Upsilon_C \in SIG(\mathcal{V})} \ \bigcup_{C \subseteq \mathcal{F}(\Upsilon, \Upsilon_C)} (\Sigma, \Upsilon, \Upsilon_C, C)$$

A *metamodeling language* is a pair (D_{meta}, τ_{meta}) where $\tau_{meta} : D_{meta} \to \Delta_{\mathcal{F}}$ maps metamodels to domains. In [1] we show how the mapping can be constructed for realistic metamodel languages. With this approach, we can extract a precise set of domain concepts and constraints from a metamodel by applying the mapping τ_{meta}. Here we overload the notation D to also represent the set of all well-formed models characterized by the domain D.

Given these preliminaries, we now turn our attention to the analysis of domains. For example, we might like to know: *Does a domain contain any non-*

[1] Technically, we should include the property that all Υ_C signatures contain $wellform(\cdot)$ or $malform(\cdot)$. We have left this out as it unnecessarily complicates the definition of $\Delta_{\mathcal{F}}$

trivial finite models?. It turns out that this fundamental question is difficult to answer for UML-like metamodels. Consider the *UNSAT* metamodel of Figure 2. If a model of *UNSAT* contains anything at all, then it contains an instance of *RootClass*. However, an instance of *RootClass* must contain at least one instance of *ClassA*, which in turn must contain at least one instance of *ClassC*. So far the constraints pose no problem. However, the inheritance operator declares that *ClassC* is a subclass of *ClassA*, so *ClassC* inherits the property that each instance must also contain at least one instance of *ClassC*. This leads to an infinite regress, so there exists no non-trivial finite model of *UNSAT*. This can be seen in Figure 3, which is a finite model that almost adheres to *UNSAT*, except that the instance *instOfC_2* does not contain another instance of *ClassC*. The degree to which we can reason about metamodels depends on the expressiveness of the constraint logic. We now turn our attention to a well-known decidable subset of first-order logic, Horn Logic.

2 Analysis of Nonrecursive Horn Domains

The simplest class of logic we examine is nonrecursive Horn logic[7]. Admittedly, this class is too small for characterizing most realistic domains, but the algorithms for manipulating this logic serve as a foundation for the more expressive logic that we describe in the next section. We begin by recalling some definitions. *Formulas* are built from terms with *variables* and logical *connectives*. There are different approaches for distinguishing variables from constants. One way is to introduce a new alphabet Σ_v that contains variable names such that $\Sigma \cap \Sigma_v = \emptyset$. The terms $T_{\Upsilon_C}(\Sigma)$ are called ground terms, and contain no variables. This set is also called the *Herbrand Universe* denoted $\mathcal{U}_H$. The set of all terms, with or without variables, is $T_{\Upsilon_C}(\Sigma \cup \Sigma_v)$, denoted $\mathcal{U}_T$. Finally, the set of all *non-ground* terms is just $\mathcal{U}_T - \mathcal{U}_H$. A *substitution* ϕ is term endomorphism $\phi : \mathcal{U}_T \rightarrow \mathcal{U}_T$ that fixes constants. In another words, if a substitution ϕ is applied to a term, then the substitution can be moved to the inside $\phi f(t_1, t_2, \ldots, t_n) = f(\phi t_1, \phi t_2, \ldots, \phi t_n)$. A substitution does not change constants, only variables, so $\forall g \in \mathcal{U}_H,\ \phi(g) = g$. We say two terms $s, t \in \mathcal{U}_T$ *unify* if there exists substitutions ϕ_s, ϕ_t that make the terms identical $\phi_s s = \phi_t t$, and of finite length. (This implies the *occurs check* is performed.) We call the pair (ϕ_s, ϕ_t) the unifier of s and t. The variables that appear in a term t are $vars(t)$, and the constants are $const(t)$.

A *Horn clause* is a formula of the form $h \Leftarrow t_1, t_2, \ldots, t_n$ where h is called the *head* and $t_1, \ldots, t_n$ are called the *tail* (or body). We write T to denote the set of all terms in the tail. The head only contains variables that appear in the tail, $vars(h) \subseteq \bigcup_i vars(t_i)$. A clause with any empty tail ($h \Leftarrow$) is called a *fact*, and contains no variables. Recall that these clauses will be used *only* to calculate model properties. This is enforced by requiring the heads to use those function symbols that do not encode model structure, i.e. every head $h = f(t_1, \ldots, t_n)$ has $f \in (\Upsilon_C - \Upsilon)$. (Proper subterms of h may use any symbol.) This is similar to restrictions placed on declarative databases[8]. We slightly extend clauses to permit *disequality* constraints. A Horn clause with disequality constraints has the form $h \Leftarrow t_1, \ldots, t_n, (s_1 \neq s'_1), (s_2 \neq s'_2), \ldots, (s_m \neq s'_m)$, where s_i, s'_i are terms with no new variables $vars(s_i), vars(s'_i) \subseteq \bigcup_i vars(t_i)$. We can now define the *meaning* of a Horn clause. The definition we present incorporates the *Closed World Assumption* which assumes all conclusions are derived from a finite initial set of facts (ground terms) I. Given a set of Horn clauses Θ, the operator $\widehat{\vdash}_\Theta$ is called the *immediate consequence operator*, and is defined as follows:

$$M \widehat{\vdash}_\Theta = M \cup \left\{ \phi(h_\theta) \mid \exists \phi, \theta,\ \phi(T_\theta) \subseteq M \texttt{ and } \forall (s_i \neq s'_i)_\theta \in \theta,\ \phi s_i \neq \phi s'_i \right\}$$

where ϕ is a substitution and θ is a clause in Θ. It can be proved that $I \vdash_\Theta I_\infty$ where $I \widehat{\vdash}_\Theta I_1 \widehat{\vdash}_\Theta \ldots \widehat{\vdash}_\Theta I_\infty$. The new terms derivable from I can be calculated by applying the immediate consequence operator until no new terms are produced (i.e. the least fixed point). Notice that the disequality constraints force the substitutions to keep certain terms distinct. *Nonrecursive Horn logic* adds the restriction that the clauses of Θ can be ordered $\theta_1, \theta_2, \ldots, \theta_k$ such that the head h_{θ_i} of clause θ_i does not unify with any tail $t \in T_{\theta_j}$ for all $j \leq i$. This is a key restriction; without it, the logic can become undecidable. Consider the recursive axiom $\Theta = \{f(f(x)) \Leftarrow f(x)\}$. Then $\{f(\mathtt{c_1})\} \vdash_\Theta \{f(\mathtt{c_1}), f(f(\mathtt{c_1})), \ldots, f(f(f(\ldots f(\mathtt{c_1}) \ldots)))\}$ includes an infinite number of distinct terms. Let $\mathcal{F}_{NH}(\Upsilon, \Upsilon_C)$ be the set of all sets of Horn clauses defined over signatures Υ, Υ_C with alphabets Σ, Σ_v.

We call domains specified with formulas from $\mathcal{F}_{NH}$ *nonrecurive Horn domains* (abbreviated NHD). The first problem we wish to solve is the membership problem for positive NHDs.

Definition 1. *The membership problem for positive NHDs: Given a positive NHD D, does there exists a finite model $M \subset \mathcal{U}_H(D)$ such that $M \vdash_C wellform(x)$ for some x. The notation $\mathcal{U}_H(D)$ indicates the set of ground terms defined by the signature Υ of D.*

The membership problem for positive NHDs is the easiest problem to solve. We will solve it by actually constructing a model M for which a $wellform(\cdot)$ term can be derived. This is possible because nonrecursive Horn logic has an important property called *monotonicity*: If a model M derives terms M', and another model N contains M, then N must derive at least M'. In symbols, $M \subseteq N$ and $M \vdash_\Theta M'$, $N \vdash_\Theta N'$, then $M' \subseteq N'$. This property implies that an algorithm only needs to examine the "smallest" models that could derive a $wellform(\cdot)$ term. Our algorithims are similar to those found in logic programming, but with some necessary augmentations. Typically, logic programs are provided with a set of initial facts that form the closed world. Our problem is to figure out the set of facts such that if the program were initialized with these facts, then the desired outcome (e.g. deriving a $wellform(\cdot)$ term) would occur. This distinction means that our algorithms cannot rely on the fact that the closed world contains a finite number of ground terms, because these terms are not yet known. It turns out that although there are an infinite number of "small" models, these models can be partitioned into a finite number of equivalence classes; these classes can be exhaustively examined.

We have developed a theorem prover called *FORMULA* (FORmal Modeling Using Logic Analysis) which implements these techniques. Figure 4 shows a positive NHD of directed graphs, called *CYCLE*, using FORMULA syntax. Line 1 declares the two function symbols v (for vertex) and e (for edge). The keyword **in** marks these as input symbols, i.e. elements of the signature Υ. The remaining symbols are used to calculate properties of an input model, and are marked **priv** for private symbols, i.e. elements of Υ_C. The theorem prover will never return a model that contains a private symbol. Well-formed models of the CYCLE

```
1:  in v arity 1; in e arity 2;
2:  priv path3 arity 3; priv path4 arity 4;
3:  priv cycle3 arity 3; priv cycle4 arity 4;
4:  priv useless arity 1; priv useless2 arity 1;
5:  priv wellform arity 1;
6:
7:  cycle3(X,Y,Z) <= path3(X,Y,Z), e(Z,X);
8:  cycle4(X,Y,Z,W) <= path4(X,Y,Z,W), e(W,X);
9:
10: path3(X,Y,Z) <= e(X,Y), e(Y,Z),
11:   !=(X,Y), !=(X,Z), !=(Z,Y);
12: path4(X,Y,Z,W) <= path3(X,Y,Z), e(Z,W),
13:   !=(W,X), !=(W,Y), !=(Y,Z);
14:
15: useless(X) <= useless2(X);
16: wellform(X) <= useless(X), e(X,Y);
17: wellform(X) <= cycle3(X,Y,Z);
18: wellform(X) <= cycle4(X,Y,Z,W);
```

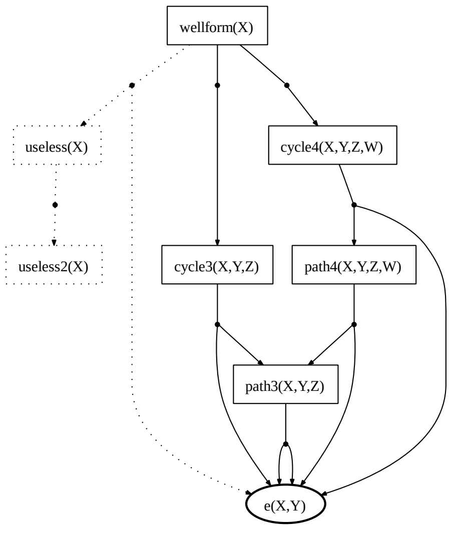

Fig. 4. (Left) CYCLE: a positive NHD in FORMULA syntax. (Right) Backwards chaining graph generated from goal $wellform(X)$.

domain must contain either a directed 3-cycle or 4-cycle. Lines 7,8 define the properties of 3-cycles and 4-cycles based on the properties of 3-paths and 4-paths. For example, a 3-cycle exists if there is a 3-path on vertices X, Y, Z and there is an edge from Z to X. (Note that the variable names are local to each clause.) Notice the use of disequality contraints in the definition of 3-paths and 4-paths in Lines 10-13. These constraints ensure that the paths contain unique vertices. Finally, Lines 16-18 define the derivation of $wellform(\cdot)$ terms.

The first step towards generating a well-formed model is to determine the derivation steps that lead to $wellform(\cdot)$ terms. This is done via an augmented form of *backwards chaining*. First, some definitions are necessary. We call two terms s, t *isomorphic* if there exists a substitution ϕ such that ϕ is a term monomorphism (one-to-one map), $\phi\ s = t$, and ϕ^{-1} is also a substitution. Clearly it holds that $s = \phi^{-1}\ t$. Given a set of terms T, let I_T be an equivalence relation on terms such that $(s, t) \in I_T$ if s and t are isomorphic. It is easy to see that I_T is an equivalence relation, because composition of monomorphisms yields another monomorphism. A *goal term* g is a term (with variables), and a solution M is a set of ground terms such $M \vdash_{\Theta} M'$ and $\exists \phi, \exists t \in M'$ $(\phi \; g = t)$. In another words, a solution is a model that derives a ground term unifying with the goal. The terms derived from the solution M are all ground terms, so, without lost of generality, it can be assumed that the unifier is $(\phi, id_{\mathcal{U}_T})$. Let $terms(D)$ be the union of all terms in the domain definition, (i.e. union of all heads and tails). Given a set of goals G and a domain D, let $[t]$ be the equivalence class of t in $I_{terms(D) \cup G}$. A *backwards chaining graph* $B(G)$ over a set of goal terms G is defined inductively as follows:

1. For each $g \in G$, $[g] \in V_{B(G)}$.
2. For all clauses $h_{\theta_i} \Leftarrow t_1, \ldots, t_m$ in Θ such that $[h_{\theta_i}] \in V_{B(G)}$, then $[t_i]_{1 \leq i \leq m} \in V_{B(G)}$ and there exists a directed "AND" edge $([h_{\theta_i}], \{[t_i]\}_{1 \leq i \leq m}) \in E_{B(G)}$.
3. For all clauses $h_{\theta_i} \Leftarrow t_1, \ldots, t_m$ in Θ such that h_{θ_i} unifies with some tail t_{θ_j} and $[t_{\theta_j}] \in V_{B(G)}$ then $[h_{\theta_i}] \in V_{B(G)}$ and there exists a directed edge $([t_{\theta_j}], [h_{\theta_i}]) \in E_{B(G)}$.

The right-hand side of Figure 4 shows the backwards chaining graph generated by the single goal term $wellform(X)$. There a significantly fewer vertices in the graph than terms in the domain definition, because many terms are isomorphic. $B(G)$ has several properties, though we will not prove them here. $B(G)$ is finite because the domain D has a finite number of clauses, and $B(G)$ is acyclic because D is nonrecursive. Unlike typical backwards chaining, the sinks in the graph are not ground terms, because their are none, but are terms with function symbols completely in Υ. Any sinks without this property are pruned from the graph. For example, the $useless(\cdot)$ and $useless2(\cdot)$ terms are pruned, because there are no ways to derive these terms from Υ terms. The vertices and edges in dotted lines are the pruned part of the graph. If a solution exists then there must be a directed path from every $[g]_{g \in G}$ vertex to a non-pruned sink using non-pruned edges. This holds because a solution contains only ground terms, which impose stronger restrictions on the unifier morphisms, than those imposed by the construction of $B(G)$.

The backwards chaining graph captures the various paths from the goal to possible solutions, and each path must be walked until a solution is found or it is confirmed that no solution exists. A path can be "unrolled" one at a time (as in SLD resolution[9]), or a tree can be constructed capturing every possible walk. We choose the latter in order to support other uses of FORMULA. The left-hand side of Figure 5 shows the unrolling of the Figure 4 into a *solution tree.* The tree has a root with a single AND edge having an endpoint on each goal term g. Every goal term g attached to the root receives an edge for each $v \in B(G)$ such that g unifies with v. For example, Figure 5 shows the vertex $wellform(V0)$ connected to the $wellform(V1)$. This edge indicates that $wellform(V0)$ unifies with $wellform(V1)$. The tree construction algorithm always *standardizes apart* unifying terms by instantiating them with unique variables. $wellform(V1)$ has two distinct paths in the backwards chaining graph, and each of these are unrolled into two subtrees of the $wellform(V1)$ vertex. If a clause has disequality constraints, then these appear as constraints on the edges in the solution tree.

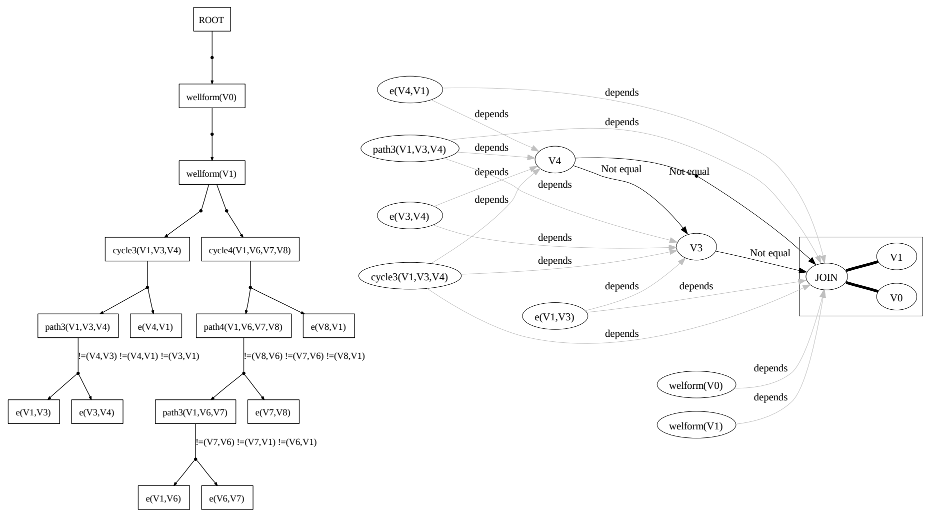

Fig. 5. (Left) Solution tree generated from backwards chaining graph of Figure 4 (Right) Constraint system shown as a forest of union-find trees.

The solution tree is viewed as a constraint system over terms. As the tree is walked, equations concerning terms are collected. A unification of terms s, t can be converted to a system of equations over variables. For example $g(X, Y)$ unifies with $g(Z, Z)$ if $X = Y = Z$. Clearly any unifier (ϕ_s, ϕ_t) must have $\phi_s(X) = \phi_s(Y) = \phi_t(Z)$. The correct equations are calculated by an inductive procedure as motivated in [9]. The constraint system is represented as a forest of union-find trees; a unification s, t yields a set of equations $\{s_i = t_i\}$, which is converted to operations on the forest: for each equation $s_i = t_i$ perform $join(find(s_i), find(t_i))$ where the $find(x)$ operation creates the vertex labeled x if x does not already exist. For example, there is one non-trivial union-find tree in Figure 5 resulting from the unification of $wellform(V0)$ with $wellform(V1)$, which joins $V0$ and $V1$. As terms are added to the forest, so are their subterms. Dependency links are maintained between vertices, where a term t is dependent on a term s if s is a subterm of t. An operation fails if the dependency edges form a cycle, essentially indicating that a multi-step unification fails. The dependency edges in Figure 5 are gray and labeled "depends". Disequality constraints are implemented as "Not equal" edges between vertices. Notice that all terms in the same union-find tree share the same constraints and dependencies. As trees are joined, all the constraints are moved up to the root. For example, in Figure 5 all constraint edges terminate on the $JOIN$ vertex. Thus, a disequality constraint fails if a vertex is unequal to itself, or a join operation moves the source and destination of a disequality edge onto the same join vertex. As the algorithm walks the solution tree, it performs operations on the constraint system. As soon as the constraint system becomes inconsistent, the algorithm restarts on an unexplored combination of subtrees. FORMULA maintains all possible restart configurations, and only fails after all restarts have been tried. Let W be the sequence of vertices visited in walk of the solution tree. Then $CS(W)$ is the constraint system produced by that walk.

After a consistent walk W has been found, the constraint system $CS(W)$ can be converted into a set of ground terms. Notice that the sinks (ignoring disequality edges) in the constraint system are those terms for which all other terms are dependent. In fact, our construction guarantees that the sinks are just variables or ground terms. Let $sinks(CS)$ be the sinks of a consistent constraint system CS defined as follows: A union-find tree $T \in CS$ is a *sink tree* if the root has no outgoing edges, or only has outgoing disequality edges. If no leaves of the sink tree are ground, then pick a leaf and place it in $sinks(CS)$. Choose any substitution ϕ_{min} such that $\phi_{min}(X) \mapsto \mathtt{c}_X \in (\Sigma - const(D))$, where $\mathtt{c}_X$ is a unique constant not appearing anywhere in the domain definition. If a variable X is in the same union-find tree as a ground term t_g, then $\phi_{min}(X) \mapsto t_g$. The values of all other variables are calculated transitively to from the full substitution ϕ_{sol}. Finally, the candidate solution M_W for walk W is

$$M_W = \left(\bigcup_{v \in W} \phi_{sol}(t_v) \right) \cap T_{\Upsilon}(\Sigma)$$

where t_v is the term of a vertex v in the walk W of the solution tree. M_W is a *proper solution* if no model terms of the form $f(t_1, \ldots, t_n)$, where $f \in \Upsilon$, are removed by the intersection with $T_{\Upsilon}(\Sigma)$. Such a term would be thrown out if it contains a subterm t_i built from symbols of $\Upsilon_C - \Upsilon$. In this case, the candidate solution is discarded and another walk through the solution tree is attempted. Applying this algorithm to the constraint system of Figure 5 gives $sinks(CS) = \{V0, V3, V4\}$. Let $\phi_{min}(V0) \mapsto \mathtt{c}_0, \phi_{min}(V3) \mapsto \mathtt{c}_1, \phi_{min}(V4) \mapsto \mathtt{c}_2$. By transitivity, $\phi_{sol}(V1) \mapsto \mathtt{c}_0$, and all variables are accounted for. Applying ϕ_{sol} to each vertex on the left-hand walk of Figure 5 gives a candidate model $M_W = \{e(\mathtt{c}_0, \mathtt{c}_1), e(\mathtt{c}_1, \mathtt{c}_2), e(\mathtt{c}_2, \mathtt{c}_0)\}$, which is a correctly constructed 3-cycle. It is not difficult to prove:

Theorem 1. *A positive NHD has a non-trivial finite model iff there exists a walk W such that $CS(W)$ is consistent and the candiate model M_W is proper.*

These algorithms can also be used to construct well-formed models with particular *embeddedings*. Let $\gamma : \mathcal{U}_H \mapsto \mathcal{U}_H$ be a term endomorphism (i.e. a homomorphism over model terms). A model M' can be *embedded* into a model M, written $M' \leq M$, if there exists a one-to-one term endomorphism (i.e. a monomorphism) such that $\gamma(M') \subseteq M$. Constructive techniques that can produce embeddings allow us to sketch a model that might be malformed, but produce a well-formed version that still contains the original model. This can be quite useful for users who do not understand all of the particular constraints of a modeling language, and would like the computer to correct mistakes. Consider the top-left graph of Figure 6. This *star* graph (S_4) is malformed with respect to the *CYCLE* domain, because it contains neither a 3-cycle nor 4-cycle. However, with a slight modification to the algorithms above, a new model can be

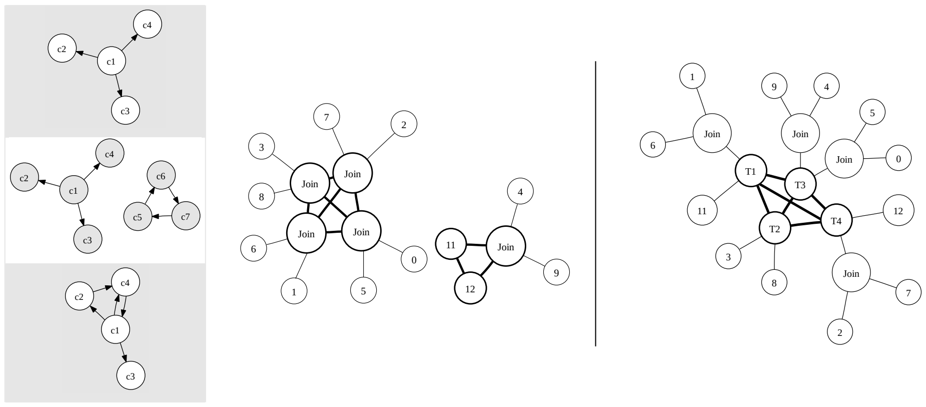

Fig. 6. (Left) A malformed input model (top), a well-formed embedding, and a minimal embedding (bottom). (Middle) Initial constraint system showing only sink trees and disequality constraints. (Right) Minimized constraint system.

built that is well-formed and contains an embeddeding of the *star* graph. Let D be a domain and let an *input model* M_I be a finite subset of model terms $T_{\Upsilon}(\Sigma)$. Choose any one-to-one map $\alpha : \Sigma \rightarrow \Sigma_v$ that uniquely relates constants to variables in Σ_v. Clearly α induces a monomorphism $\phi_\alpha : T_{\Upsilon}(\Sigma) \rightarrow T_{\Upsilon}(\Sigma_v)$ from terms without variables to terms that only have variables. We will use this monomorphism to encode the input model as a Horn clause. Pick any function symbol $f \notin \Upsilon_C$ and add it Υ_C with arity $|consts(M_I)|$, i.e. the arity of f is equal to the number of constants in the input model M_I. Add the following clause θ_{M_I} to D:

$$\theta_{M_I} \doteq f(\alpha(c_1), \alpha(c_2), \ldots, \alpha(c_n)) \Leftarrow \bigwedge_{t_m \in M_I} \phi_\alpha(t_m) \bigwedge_{i \neq j} (\alpha(c_i) \neq \alpha(c_j))$$

where $1 \leq i, j \leq n = |consts(M_I)|$. Recall from the previous algorithms, that a solution is constructed by defining a substitution ϕ_{sol} that is determined by the sink variables $sinks(CS(W))$. Consider any solution to any goal set G where $f(\alpha(c_1), \ldots, \alpha(c_n)) \in G$. By construction, the restriction of ϕ_{sol} to $sinks(CS(W))$ yields a one-to-one map. In the construction above, all pairs of variables induced by M_I have disequality constraints, so $\alpha(consts(M_I)) \subseteq sinks(CS(W))$ for any consistent walk W^2. Therefore, the restriction of any ϕ_{sol} to the terms $T_{\Upsilon}(\alpha(consts(M_I)))$ must be a monomorphism. Thus, $\gamma = (\phi_{sol} \circ \phi_\alpha)$ gives the embeddeding of M_I in any proper solution M_W for a consistent walk W.

Theorem 2. *Given an input model M_I and a positive NHD D, augmented with f and θ_{M_I}. Any proper solution to a goal set G, where $f(\alpha(c_1), \ldots, \alpha(c_n)) \in G$, contains an embeddeding of M_I.*

[2] This is a slight simplification. There will be some representative sink variable for each variable in the image of α.

In particular, let the goal set $G = \{f(\alpha(c_1), \ldots, \alpha(c_n)), wellform(X)\}$, where the variable X is not in image of α, then any solution to G contains M_I and is well-formed. The middle-left graph of Figure 6 shows FORMULA's construction of a well-formed version of the star graph in the *CYCLE* domain.

The default embedding produced by FORMULA is not particularly elegant. It contains a star juxtaposed with a 3-cycle. This solution was constructed because ϕ_{sol} assigns a unique constant to each sink variable, yielding a *maximal solution* with respect to the number of constants. A smaller solution can be found by manipulating the final constraint system $CS(W)$ so that the number of sink variables are reduced. This can be accomplished by merging sink trees, which is legal if the trees do not have disequality constraints between them. The middle graph of Figure 6 shows the sink trees of the constraint system after producing the middle-left embeddeding. The root of each tree is in bold, and disequality constraints between trees are shown as bold edges. These are the only types of edges between trees, because sink trees do not have dependency edges between them. A *minimal solution* can be formed by partitioning the root vertices into a minimal number of independent sets. This is a computationally hard optimization problem related to the *independent set problem*. The right side of Figure 6 shows the optimized constraint system, which contains only four trees (and four sink variables). The roots of the optimized constraint system form a clique, therefore no further optimization is possible. The bottom-left graph shows the optimized solution generated by FORMULA, wherein the star and 3-cycle have been merged in an ideal fashion. Note that this process yields a minimal, but not neccessarily *minimum* model. Finding a minimum model requires minimizing all possible consistent walks of the solution tree.

3 Extensions, Tools, and Future Directions

We have shown that the constructive reasoning of UML-like metamodels is a rich area of study, both theoretically and algorithmically. In the interest of space we have used directed graphs as our toy example. However, these techniques can be applied to much more complicated metamodels, and with practical applications: *Metamodel composition* is the process of constructing new domain-specific languages by combining existing metamodels. Two metamodels mm_1 and mm_2, can be syntactically combined with an operator $\circ$, such as class equivalence[10], and the syntactic composition can be converted into a domain $D_{comp} = \tau_{meta}(mm_1 \circ mm_2)$. The membership problem for the domain can then be solved, thereby deciding if the metamodel composition is *semantically* meaningful. Other problems, like the construction of embeddedings, correspond to the automatic construction of useful models that satisfy the domain constraints. Model transformations can also be incorporated into our framework, and then constructive techniques can be used to prove that the transformation always produces well-formed output models from well-formed input models. This is the weakest form of correctness one could imagine, but checking these properties has remained mostly open. There is already precedent for the use of Prolog engines to transform a particular input model M_I to an output model M_O, as is done by Viatra2[11]. A particular input/output pair (M_I, M_O) can be compared to check for mutual consistency (e.g. via bisimulation). However, checking properties of the overall transformation is more difficult, though our approach can handle it as long as the transformation is restricted to an appropriate class of logic. The verification goal resembles Hoare's notion of a *verifying compiler*[12].

This brings us to questions of expressiveness. How expressive is Horn logic and how far can it be taken? This question has driven our development of FORMULA, which we now summarize. Positive NHDs are not particularly expressive, but they are an essential starting point for developing constructive techniques for more expressive domains. The next step in the progression is to solve the membership problem for *negative* NHDs. Recall that negative domains characterize the malformed models with the symbol $malform(\cdot)$, and a model M is wellformed if $\forall x, M \nvdash_C malform(x)$. Negative NHDs can express domains not representable by positive NHDs, because of the universal quantification over $malform(x)$. Notice that the solution tree for a goal $G = \{malform(x)\}$ contains all equivalence classes of malformed models, and the *malformed-ness* property is monotonic in models. With these observations, the membership problem can be solved by repeating this procedure: Prune all leaves in $B(\{malform(x)\})$, except for one symbol $f \in \Upsilon$. If $malform(x)$ can be proved on the corresponding pruned solution tree, then by monotonicity, no wellformed model can contain a term unifiying with f. If $malform(x)$ cannot be proved, then a wellformed model $M = \{f(\cdot, \ldots, \cdot)\}$ has been found. This test is repeated (at least once) for each $f \in \Upsilon$; due to unification issues, it may be repeated multiple times for non-unifying f-terms. This procedure is also implemented in FORMULA.

A further increase in expressiveness can be obtained by extending the Horn logic so that a tail can contain a "negated" term $\neg t_i$. (For example, the *UNSAT* domain (Figure 2) can be defined with this extension.) Loosely, a negated term is a constraint requiring that a solution $M \nvdash_C t_i$. Theoretically, this extension approximates the power of full first order logic, but remains decidable (under additional restrictions on its use). It turns out that this simple extension corresponds to a *nonmonotonic* logic, and has deep theoretical and algorithmic repercussions. Our major challenge has been the development of constructive techniques for domains written in Horn logic extended with negation. These techniques are also implemented in FORMULA, and extend existing work on nonmonotonic inference[7][13] to deal with the particulars of UML-like metamodels. Theoretically, these extensions must be handled carefully in order to maintain the soundness and completeness of the theorem prover. Algorthmically, our approach combines the aforementioned algorithms with state-of-the-art SAT solvers to construct models. In conclusion, a reasonable level of expressiveness can be obtained.

A common criticism of theorem proving is the requirement of the user to understand the underlying mathematics. We have addressed this issue by developing an automated conversion from metamodels to domain definitions. This approach is described in [1], and supports metamodeling in the well-known

Generic Modeling Environment (GME) toolsuite[14]. Furthermore, because the theorem prover is constructive, the results of the prover are concrete models that can be automatically imported back into the GME modeling environment. This closes the loop, providing constructive reasoning about models and metamodels without leaving the comfort of the modeling toolsuite (for most of the common queries). Our future work is to apply these techniques to analyze model transformations, including those specified with the Graph Rewriting and Transformation (GReAT) language[15] that is also part of the GME toolsuite.

References

1. Jackson, E.K., Sztipanovits, J.: Towards a formal foundation for domain specific modeling languages. Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT'06) (October 2006) 53–62
2. Object Management Group: Meta object facility specification v1.4. Technical report (2002)
3. Object Management Group: Unified modeling language: Superstructure version 2.0, 3rd revised submission to omg rfp. Technical report (2003)
4. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. J. Comput. Syst. Sci. **64**(2) (2002) 249–283
5. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: ICGT. (2004) 161–177
6. Burris, S.N., Sankappanavar, H.: A Course in Universal Algebra. Springer-Verlag (1981)
7. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3) (2001) 374–425
8. Minker, J.: Logic and databases: A 20 year retrospective. In: Logic in Databases. (1996) 3–57
9. Chan, D.: Constructive negation based on the complete database. In: Proc. Int. Conference on LP'88, The MIT Press (1988) 111–125
10. Emerson, M., Sztipanovits, J.: Techniques for metamodel composition. OOPSLA 2006 Domain-Specific Languages Workshop (2006)
11. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: Viatra: Visual automated transformations for formal verification and validation of uml models (2002)
12. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1) (2003) 63–69
13. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080
14. G. Karsai, J. Sztipanovits, A.L.T.B.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (January 2003) 145–164
15. G. Karsai, A. Agrawal, F.S.: On the use of graph transformations for the formal specification of model interpreters. Journal of Universal Computer Science **9**(11) (November 2003) 1296–1321