Model Generation for Horn Logic with Stratified Negation

Ethan K. Jackson and Wolfram Schulte

Microsoft Research,
One Microsoft Way, Redmond, WA
{ejackson,schulte}@microsoft.com

Abstract. Model generation is an important formal technique for finding interesting instances of computationally hard problems. In this paper we study model generation over Horn logic under the closed world assumption extended with stratified negation. We provide a novel three-stage algorithm that solves this problem: First, we reduce the relevant Horn clauses to a set of non-monotonic predicates. Second, we apply a fixed-point procedure to these predicates that reveals candidate solutions to the model generation problem. Third, we encode these candidates into a satisfiability problem that is evaluated with a state-of-the-art SMT solver. Our algorithm is implemented, and has been successfully applied to key problems arising in model-based design.

1 Introduction

Informally, *model generation* is a procedure that takes as input some mathematical statement ψ, and produces as output some data M (a model) that, when substituted back into ψ, makes the statement true. For example, if ψ is a boolean satisfiability problem, then M is an assignment of boolean variables to truth values. Similarly, if ψ is a set of linear inequalities, then M is an assignment of variables to the real numbers. Note that model generation can be used to check satisfiability, but not all techniques for checking satisfiability are able to generate models. In this paper we study model generation for an important type of non-classical logic called *Horn logic with stratified negation*.

Horn logic has important applications in computer science and new applications continue to arise. Recently, Horn logic extended with *negation as failure* was used to formalize the non-context-free languages arising in modern software engineering methodologies[1] such as *Model Driven Architecture*[2][3] (MDA), *Model Integrated Computing*[4] (MIC), and *Platform-based Design*[5][6] (PBD). This particular application adds a new and interesting twist: An effective means of model generation is essential if the Horn paradigm is to be truly useful.

In this paper we present a novel approach to model generation for *non-recursive Horn logic* extended with *stratified negation*. Our approach employs a three stage process:

1. We simplify the problem by reducing the relevant Horn clauses to a set of non-monotonic predicates that we call *non-monotonic acceptors*.

2. We apply a fixed-point procedure to the non-monotonic acceptors that reveals the candidate solutions to the model generation problem.
3. We encode these candidates into a satisfiability problem that is evaluated with the state-of-the-art SMT solver Z3.

We show that this procedure is sound, but incomplete.

This paper is organized into six sections. Section 2 informally describes the class of Horn logic targeted for model generation. Section 3 provides the key formal definitions. The first stage of the algorithm is explained in Section 4 as a modified form of backwards chaining. Section 5 describes the elimination of quantification over closed-worlds and the reduction to a boolean satisfiability problem. We conclude in Section 6.

2 Background and Running Example

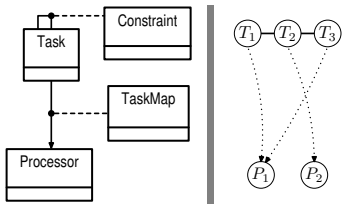

Fig. 1. (Left) Scheduling abstraction (Right) Example scheduling instance

Model-based approaches to software engineering rely on high-levels of abstraction to simplify the design process. The left-hand side of Figure 1 shows a *metamodel* describing a simple abstraction layer. This diagram defines an abstract language for scheduling problems without regard to the particular details of the tasks being scheduled. This language contains objects of type `Task` and `Processor`. Tasks can be assigned to processors by directed edges of type `TaskMap`. Resource constraints between tasks prevent two tasks from being scheduled on the same processor. Resource constraints are modeled as undirected edges of type `Constraint` connecting tasks. The right-hand side of Figure 1 shows a member of this language. There are three tasks T_1, T_2, T_3 and two processors P_1, P_2. Tasks T_1 and T_2 have a resource constraint, as do tasks T_2 and T_3. Tasks T_1, T_3 are scheduled on processor P_1 while T_2 is scheduled on P_2.

Metamodels (and other artifacts) can expressed a set of axioms using *Horn logic with stratified negation* [7]. Consequently, model generation on this logic is a key tool for reasoning about abstraction layers. For example, we might demand a model generator to *"Construct a model that contains three tasks and two processors"*. The procedure must find a model satisfying the rules of the abstraction that also meets this goal. Model generation is a difficult problem, as this example illustrates. A correct mapping from tasks to processors is precisely a graph

coloring problem[8] where the processors are the colors. This illustrates that any procedure capable of constructing non-trivial instances must solve difficult subproblems.

The task scheduling language is defined with the following set of non-recursive and stratified Horn clauses:

$$task(x) \leftarrow taskmap(x, y) \tag{1}$$
$$processor(y) \leftarrow taskmap(x, y) \tag{2}$$
$$task(x) \leftarrow constraint(x, y) \tag{3}$$
$$task(y) \leftarrow constraint(x, y) \tag{4}$$
$$no\_map(task(x)) \leftarrow task(x), \neg taskmap(x, y) \tag{5}$$
$$bad\_map(task(x), task(y)) \leftarrow taskmap(x, z), taskmap(y, z), constraint(x, y) \tag{6}$$

The first four clauses declare that the end-points of task mappings and resource constraints always exist. Clause 5 deduces a term $no\_map(task(x))$ for any task x that is not mapped to a processor. Clause 6 deduces $bad\_map(task(x), task(y))$ anytime two tasks x, y are improperly scheduled. We now examine the semantics of this logic in detail.

2.1 Classical Horn Logic

Classical Horn logic restricts first-order logic by requiring each conjunct of a DNF (disjunctive normal form) formula ψ to have at most one non-negated literal. A collection of Horn formulas has a more natural representation in *implicative normal form*, as shown below:

$$\forall x, y, z\ taskmap(x, z), taskmap(y, z), constraint(x, y) \Rightarrow bad\_sched(x, y) \tag{7}$$

This classical clause looks similar to Clause 6, however its meaning is quite different. For the sake of discussion, assume that $taskmap(\cdot, \cdot)$, $constraint(\cdot, \cdot)$, $bad\_sched(\cdot, \cdot)$ are predicates. From this clause we know that $bad\_sched(x, y)$ must be true for tasks x and y that have resource constraint between them and are scheduled onto the same processor. Given two particular tasks $\mathtt{t}_1$, $\mathtt{t}_2$ without a resource constraint between them, what can we conclude about $bad\_sched(\mathtt{t}_1, \mathtt{t}_2)$? Rewriting Equation 7 yields:

$$\forall z,\ \neg taskmap(\mathtt{t}_1, z) \vee \neg taskmap(\mathtt{t}_2, z) \vee \neg constraint(\mathtt{t}_1, \mathtt{t}_2) \vee bad\_sched(\mathtt{t}_1, \mathtt{t}_2) \tag{8}$$

Since $\neg constraint(\mathtt{t}_1, \mathtt{t}_2)$ is true, Equation 8 is satisfied without forcing a particular truth value to $bad\_sched(\mathtt{t}_1, \mathtt{t}_2)$. In other words, there exists a model satisfying Equation 7 for which tasks 1 and 2 are badly scheduled, but there also exists a model where they are not badly scheduled. Both of these possibilities exist because we have used classical implication. However, there exists a commonly employed extension to Horn logic that closes this loop-hole.

2.2 Closed World Assumption

The *Closed World Assumption* (CWA) is applied whenever a set of Horn clauses is intended to capture *all* the necessary information for a domain[9]. In order for CWA to work properly there must exist some information known to be true from the outset. These pieces of information are called *facts*, which are of the form $true \Rightarrow h$ where h is a non-negated literal. For example:

$$true \Rightarrow taskmap(\mathtt{t}_1, \mathtt{p}_1) \tag{9}$$

Intuitively, a predicate $f(x, y, z)$ is true for some x, y, z if $f(x, y, z)$ is a fact or if there is a sequence of derivations starting at facts that force $f(x, y, z)$ to be true. If there is no such derivation, then $f(x, y, z)$ is false. This rule eliminates the case where tasks $\mathtt{t}_1$ and $\mathtt{t}_2$ are badly scheduled.

This slight adjustment to classical implication profoundly effects the underlying formal machinery by introducing a fixed-point operator $\widehat{\Gamma}$. This operator, called the *immediate consequence operator*, deduces new facts from existing facts using the clauses. All facts not deduced by $\widehat{\Gamma}$ are false, so any model generation procedure must reason over this operator. If M is an initial set of facts and Λ is a set of non-fact Horn clauses, then the set of facts deducible by Λ is the least set X such that $M \subseteq X$ and $X = \widehat{\Gamma}(X)$. CWA can also be understood from a different angle under the name *existential fixed-point logic* [10].

The Closed World Assumption is used in most applications of Horn logic, so it must be taken into account by any model generation procedure. However, CWA forces a rephrasing of the model generation problem: Let Λ be a set of non-fact Horn clauses and a *goal* $G = \{l_1, l_2, \ldots, l_n\}$ be a set of non-negated literals. Loosely speaking, the *finite* model generation problem is to find a finite set of facts M so that (1) $M \subseteq X$, (2) X is a least fixed-point of $\widehat{\Gamma}$, and (3) X contains the goal literals (with respect to some substitution.) Thus, any model generation procedure must reason carefully about the fixed-points of $\widehat{\Gamma}$.

2.3 Negation-as-Failure

Classical Horn logic (without CWA) restricts the use of negation, which restricts the expressiveness of the fragment. *Negation-as-failure* (NAF) attempts to reintroduce a form of negation that is compatible with CWA and does not recreate full first-order logic. However, this new form of negation is very different from its classical counterpart. Intuitively, a negated literal $\neg l$ is true if l *cannot* be proved true under Horn logic with CWA. Thus, negation is defined in terms of a proof procedure.

In order to distinguish our Horn clauses from the classical fragment we write a clause this way:

$$h \leftarrow s_1, s_2, \ldots, s_m, \quad \neg t_1, \neg t_2, \ldots, \neg t_k \tag{10}$$

The literal h is called the *head* of the clause and $\{s_1, \ldots, s_m, t_1, \ldots, t_k\}$ is the *tail* of the clause. Each $\neg t_i$ is a *negated* literal where negation refers to non-classical NAF. Consider Clause 5 containing the negated term $\neg taskmap(x, y)$.

This negation does not directly ask if $taskmap(x, y)$ is false for some x, y. Instead, it asks if $taskmap(x, y) \notin \widehat{\Gamma}(M)$ for some x, y. Unlike classical negation, NAF must be used carefully otherwise logical inconsistencies can arise. For example, under NAF we might simultaneously conclude $f \in \widehat{\Gamma}(M) \wedge f \notin \widehat{\Gamma}(M)$ for some fact f. This is a more dangerous inconsistency than $b \wedge \neg b$ for some boolean variable b, which has a well-defined meaning. Much work has been done on generalized forms of NAF that do not suffer from inconsistencies [11][7][12]. We avoid these problems by using only a restricted form of NAF called *stratified negation*. Our approach to handling NAF is similar to the *non-monotonic rules* described in [13].

3 Definitions

We now formally describe the style of Horn logic for which we generate models; this logic incorporates both CWA and NAF. Note that our definitions are biased to make the presentation of model generation simpler.

3.1 Basic Concepts

Let Υ denote a finite signature, Σ an infinite alphabet of constants, and $\mathcal{V}$ a infinite alphabet of variable names. We use the letters f, g, h for variables ranging over function symbols of some signature Υ. We use `typewriter` script to denote constants from Σ. Finally, we use x, y, z for variables ranging over *terms*. Let $arity(f)$ denote the arity of some function symbol f. A *term* is a combination of function symbols, constants, and variables:

Definition 1. *Given Υ, Σ, and $\mathcal{V}$, the set of all finite terms $\mathcal{T}$ is defined inductively*

1. *Each $\mathtt{c} \in \Sigma$ is a term*
2. *Each $x \in \mathcal{V}$ is a term*
3. *If $f \in \Upsilon$ and $t_1, t_2, \ldots, t_{arity(f)} \in \mathcal{T}$ then $f(t_1, t_2, \ldots, t_{arity(f)})$ is a term.*

If it is unclear from context, we write $\mathcal{T}(\Upsilon)$ to denote the finite terms constructed from function symbols of signature Υ. A *ground term* is a term without variables; we use $\mathcal{T}_G$ to denote the set of all ground terms. If t is a term, then s is a *subterm* of t (written $s \sqsubseteq t$) if $s = t$ or s is a subterm of one of the arguments of t. If t is a term, then $vars(t)$ is the set of subterms that are also variable names. Similarly, $consts(t)$ is the set of subterms that are also constants. These functions are extended to sets of terms $S \subseteq \mathcal{T}$ in the natural way: $vars(S)$ (or $consts(S)$) are the variables (or constants) appearing in a set of terms.

3.2 Substitutions and Unifiers

Terms are related to one another through special homomorphisms called *substitutions*.

Definition 2. *A substitution $\varphi : \mathcal{T} \to \mathcal{T}$ is a mapping from terms to terms such that:*

1. *φ fixes constants, i.e. $\forall \mathtt{c} \in \Sigma,\ \varphi(\mathtt{c}) = \mathtt{c}$.*
2. *φ is a term homomorphism, i.e. $\varphi(f(t_1, t_2, \ldots, t_n)) = f(\varphi(t_1), \varphi(t_2), \ldots, \varphi(t_n))$.*

Let Φ be the set of all substitutions for some $\mathcal{T}$. Two terms s, t are said to *unify* if there exists a substitution φ that makes them the same: $\varphi(s) = \varphi(t)$. Essentially, a substitution makes two terms the same by replacing variables in the terms with new subterms. The essence of this replacement is easily characterized in terms of the *kernel* of φ.

Definition 3. *The kernel of a homomorphism φ (e.g. a substitution) is:*

$$\mathbf{ker}\ \varphi = \left\{(s,t) \in \mathcal{T}^2 \mid \varphi(s) = \varphi(t)\right\} \tag{11}$$

The kernel characterizes which subterms are equated by a substitution without regard to the particular values assigned to variables. Some important properties of kernels are: (1) Every kernel is an equivalence relation. (2) The intersection of two equivalence relations is also an equivalence relation. (3) The least equivalence relation Θ containing two equivalence relations Θ_1, Θ_2 is the transitive closure of $(\Theta_1 \cup \Theta_2)$. This shall be written $\Theta = \Theta_1 \oplus \Theta_2$.

The *most general unifiers* (mgu) of two terms s, t is an equivalence relation between the variables of s and t that must hold for any substitution unifying s and t. This equivalence relation represents the weakest set of constraints over the variables of s and t that ensures unification. The most general unifiers have the following properties:

Lemma 1. *Given two terms s, t that unify, let $mgu(s,t)$ denote the most general unifiers.*

1. *The $mgu(s,t)$ is unique.*
2. *$mgu(s,t) = \bigcap_{\varphi} \{\mathbf{ker}\ \varphi \mid \varphi(s) = \varphi(t)\}$*

If terms s and t do not unify, then we write $mgu(s,t) = \emptyset$.

3.3 Horn Logic with CWA and NAF

Given $\Upsilon, \Sigma, \mathcal{V}$ a Horn clause λ is a triple $\lambda = (h, P, N)$ where h is a term and P, N are sets of terms. $P = \{s_1, \ldots, s_m\}$ is the set of non-negated tail terms and $N = \{t_1, \ldots, t_k\}$ is the set of negated tail terms. A Horn clause is written:

$$h \leftarrow s_1, s_2, \ldots, s_m, \quad \neg t_1, \neg t_2, \ldots, \neg t_k \tag{12}$$

Furthermore, P must be non-empty and $vars(h) \subseteq vars(P)$. (We shall explain these restrictions shortly.) Let Λ be a finite set of Horn clauses, then there exists a binary relation $\prec$ over clauses, where $(h', P', N') \prec (h, P, N)$ if there exists some $s_i \in P$ or $t_j \in N$ that unifies with h'.

Definition 4. *Let Λ be a finite set of clauses, then Λ is* non-recursive and stratified *if $\prec$ is a strict partial order.*

Restricting $\prec$ to a strict partial order yields a simple semantics for evaluating the set of facts derivable by Λ. Order the clauses $\lambda_1, \lambda_2, \ldots, \lambda_k$ so that $\lambda_i \prec \lambda_j$ implies $i < j$, then for each clause define an immediate consequence operator:

$$\widehat{\Gamma}_i(X) = \bigcup_{\varphi} \left\{ \varphi(h_i) \mid \alpha(P_i, N_i, \varphi, X) \right\} \cup X \tag{13}$$

This equation states that the facts deducible by a single clause λ_i are calculated by finding all the substitutions that satisfy a special predicate $\alpha_i(P_i, N_i, \varphi, X)$; each substitution is applied to the head h_i to derive a new fact. Earlier we restricted the variables of h_i to be a subset of the variables of P_i, so each substitution maps h_i to a well-defined ground term.

The predicate $\alpha(P_i, N_i, \varphi, X)$ captures the CWA and NAF semantics.

Definition 5. $\alpha : \mathcal{P}(\mathcal{T})^2 \times \Phi \times \mathcal{P}(\mathcal{T}) \rightarrow \mathbb{B}$ *is called a* non-monotonic acceptor:

$$\alpha(P, N, \varphi, X) \stackrel{def}{=} (\varphi(P) \subseteq X) \wedge \forall \varphi' \left[\big(\varphi(P) = \varphi'(P)\big) \Rightarrow \big(\varphi'(N) \cap X = \emptyset\big) \right] \tag{14}$$

The acceptor $\alpha(P, N, \varphi, X)$ is true for some substitution φ and some set of terms (e.g. facts) X if the positive terms P can be found in the set of facts through the substitution φ. The negative terms N must not be found in the facts X for any extension of φ to φ' that agrees on P.

Lemma 2. *Let Λ be a finite set of non-recursive and stratified Horn clauses, and M a finite set of ground terms. Let the clauses of Λ be ordered $\lambda_1, \lambda_2, \ldots, \lambda_k$ to respect $\prec$ then:*

1. *The set of all facts deducible from M by Λ is $\Gamma(M)$ where:*

$$\Gamma(M) = \widehat{\Gamma}_k(\ldots \widehat{\Gamma}_2(\widehat{\Gamma}_1(M)) \ldots) \tag{15}$$

2. *It can be decided in finite time if any ground term $t_g \in \Gamma(M)$, i.e. the logic is decidable.*

3.4 The Model Generation Problem

Solving the model generation problem requires the construction of a set of facts M that satisfies a *goal*. A goal $G = (P_G, N_G)$ is comprised of two sets of terms: the positive terms P_G, and the negative terms N_G. A goal is satisfied if there exists some M such that all the facts deduced from M (i.e., $\Gamma(M)$) include P_G and do not include N_G. More precisely, M satisfies the goal if $\exists \varphi,\ \alpha(P_G, N_G, \varphi, \Gamma(M))$ holds.

In order to construct meaningful solutions, the model generation procedure must know which terms are allowed to appear as facts. Consider the problem of creating a badly scheduled set of tasks for the scheduling abstraction:

$$G = (\{bad\_map(x, y)\}, \emptyset)$$

then we expect the model generation procedure return a solution similar to this one:

$$M = \left\{ \begin{array}{c} task(\mathtt{t}_1), task(\mathtt{t}_2), processor(\mathtt{p}_1), \\ taskmap(\mathtt{t}_1, \mathtt{p}_1), taskmap(\mathtt{t}_2, \mathtt{p}_1), constraint(\mathtt{t}_1, \mathtt{t}_2) \end{array} \right\}.$$

Without additional information, the solution $M = \{bad\_map(\mathtt{c}, \mathtt{c})\}$ is also trivially valid. This extra information is expressed by partitioning the signature Υ into two parts: the *fact signature* Υ_F and the *derived signature* Υ_D. We call a term $t \in \mathcal{T}(\Upsilon_F)$ a *fact term* and call all other terms *derived terms*. The model generation procedure only considers solutions that are sets of fact terms. For example, the partitioning:

$$\begin{array}{l} \Upsilon_F = \{task(\cdot), processor(\cdot), taskmap(\cdot,\cdot), constraint(\cdot,\cdot)\}, \\ \Upsilon_D = \{no\_map(\cdot), bad\_map(\cdot,\cdot)\} \end{array}$$

forces all solutions to be built from tasks and processors. The goal G contains a derived term $bad\_map(x, y)$, but the solution M will never contain this term directly. We now define the finite model generation problem.

Definition 6. *The finite model generation problem - Given:*

1. *A finite signature Υ partitioned into $\Upsilon_F \neq \emptyset$ and Υ_D,*
2. *A finite set of clauses Λ that are non-recursive and stratified,*
3. *A goal $G = (P_G, N_G)$ where P_G, N_G are finite subsets of terms.*

Construct a finite set of ground terms $M \subset \mathcal{T}_G(\Upsilon_F)$ so that

$$\exists \varphi \ \alpha(P_G, N_G, \varphi, \Gamma(M)) \tag{16}$$

4 Utilizing Backwards Chaining

The semantics of non-recursive and stratified Horn logic has a succinct characterization in terms of Γ. However, it is difficult to construct a set of constraints from Γ that guide model generation. Fortunately, there is a well-known technique for computing the truth values of goals, called *backwards chaining*, which addresses this problem. Imagine that a set of facts M is already known, and the only task is to check if a goal G is satisfied by these facts. Backwards chaining works backwards from the goal, through the clauses, to the facts M to check satisfiability. If the goal is satisfied, then the procedure yields a proof tree showing exactly how the facts derive the goal. Formally, this process is called *SLD resolution* [14] for Horn logic and *SLDNF resolution* [15] for Horn logic with NAF. These resolution procedures are sound and complete for non-recursive and stratified Horn logic. We modify SLDNF resolution to return "possible" proof trees, and then search for models that satisfy these proof trees. We utilize soundness/completeness results [16] to argue soundness for model generation.

The key modification to SLDNF is a new termination condition that does not rely on M. Typical backwards chaining terminates when it encounters a fact, i.e. a clause of the form $f \leftarrow true$. This termination condition must be modified

for model generation because initially no facts are known. We modify backwards chaining so that it terminates when a fact term is encountered, even though this fact term may not exist in the solution M. Let a clause $\lambda \in \Lambda$ be partitioned as follows:

$$h \leftarrow p_1, \ldots, p_m, \quad u_1, \ldots, u_{m'}, \quad \neg n_1, \ldots, \neg n_k, \quad \neg w_1, \ldots, \neg w_{k'} \tag{17}$$

where (1) each p_i is a positive fact term, (2) each u_i is a positive derived term, (3) each n_i is a negative fact term, and (4) each w_i is a negative derived term.

Associated with each clause λ is a backwards chaining predicate $\beta_\lambda(\varphi, M, \Theta)$ where M is a set of terms, φ is a substitution, and Θ is an equivalence relation.

Definition 7. *Let λ be a clause, then associate with λ a backwards chaining predicate*[1] *$\beta_\lambda(\varphi, M, \Theta)$:*
$\beta_\lambda(\varphi, M, \Theta) \stackrel{def}{=}$

1. $\big(\Theta \subseteq \mathbf{ker}\ \varphi\big) \wedge$
2. $\alpha\Big(\{p_1, p_2, \ldots, p_k\}, \{n_1, n_2, \ldots, n_m\}, \varphi, M\Big) \wedge$
3. $\displaystyle\bigwedge_{1 \le i \le m'} \left(\bigvee_{mgu(u_i, h_{\lambda'}) \neq \emptyset} \beta_{\lambda'}\Big(\varphi, M, \Theta \oplus mgu(u_i, h_{\lambda'})\Big) \right) \wedge$
4. $\displaystyle\forall_{\varphi'} \left[\begin{array}{l} \Big(\forall v \in vars(P_\lambda)\ \varphi'(v) = \varphi(v)\Big) \Rightarrow \\ \left(\bigwedge_{1 \le j \le k'} \bigwedge_{mgu(w_j, h_{\lambda''}) \neq \emptyset} \neg\beta_{\lambda''}\Big(\varphi', M, \Theta \oplus mgu(w_j, h_{\lambda''})\Big) \right) \end{array} \right]$

The backwards chaining predicate is defined recursively and terminates on fact terms; these parts of the tail simplify to non-monotonic acceptors (Def. 7.2). On the other hand, derived terms must be understood through additional clauses. The backwards chaining process recurses into derived terms by locating clauses that unify with these terms. The equivalence relation Θ is used to collect unification constraints during this process. For every positive derived u_i there must exist some unifying clause λ' so that $\beta_{\lambda'}$ is satisfied (Def. 7.3). Contrarily, every negative derived term w_j must have no clause λ'' that derives w_j for any extension of φ to φ' agreeing on positive variables (Def. 7.4). It is possible that some unification constraints cannot be satisfied by any substitution φ. If this occurs then Def. 7.1 fails to hold. We assume that each time a clause λ' (or λ'') is examined for unification its variables are renamed to new variables that have not appeared before. This is called *standardizing apart*, and it prevents clauses from improperly interacting through variable names.

Backwards chaining is used to reduce any goal into a set of non-monotonic acceptors. However, there is one problem with the simple definition presented

[1] We follow the convention that the OR of the empty set is false and the AND of the empty set is true.

here: It does not recurse through clauses with heads that are fact terms. (Consider clause 1 from the previous example.) This definition assumes that fact terms do not appear as heads. Fortunately, we can always rewrite the clauses of Λ to enforce this rule; this is discussed later. For the moment, assume that facts do not appear as heads then the following important theorem holds:

Theorem 1. *Given Υ_F, Υ_R, Λ such that fact terms do not appear as heads, and a goal G, then*

$$\forall M \subset \mathcal{T}(\Upsilon_F),\ \forall \varphi \quad \alpha\big(P_G, N_G, \varphi, \Gamma(M)\big) \Leftrightarrow \beta_G(\varphi, M, \mathcal{ID}_{\mathcal{T}}) \tag{18}$$

for M finite and Λ non-recursive and stratified.

This theorem shows that evaluating the non-monotonic acceptor over all the facts deducible from Γ gives the same result as working backwards from the goal G through the backwards chaining predicates. We use this result to eliminate the fixed-point operator Γ from the model generation problem. Note that the backwards chaining process is initiated without any constraints on the variables as described by the identity relation $\mathcal{ID}_{\mathcal{T}} = \{(t,t) | t \in \mathcal{T}\}$.

4.1 Simplification of Backwards Chaining

The backwards chaining formulation eliminates Γ, but generates many constraints across many recursions. In this section we show how to aggregate these constraints into convenient pieces. To facilitate this discussion we give names to particular parts of the backwards chaining predicate:

$$\omega(\varphi, \varphi', P) \stackrel{def}{=} \Big(\forall v \in vars(P)\ \varphi'(v) = \varphi(v)\Big) \tag{19}$$

$$\psi^-_\lambda(\varphi', M, \Theta) \stackrel{def}{=} \bigwedge_{1 \le j \le k'} \bigwedge_{mgu(w_j, h_{\lambda''}) \neq \emptyset} \neg\beta_{\lambda''}\Big(\varphi', M, \Theta \oplus mgu(w_j, h_{\lambda''})\Big) \tag{20}$$

Definition 7.4 becomes $\forall \varphi'\ \omega(\varphi, \varphi', P_\lambda) \Rightarrow \psi^-_\lambda(\varphi', M, \Theta)$.

Consider the action of the goal backwards chaining predicate $\beta_G(\varphi, M, \Theta)$, as shown in Figure 2. The predicate β_G introduces a non-monotonic acceptor α_1 and some constraints on the kernel of φ via Θ_1. (Note that we index the

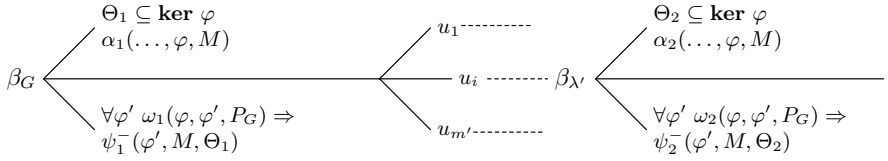

Fig. 2. A single expansion of the backwards chaining predicate for some unification choices of positive derived terms.

constraints $1, 2, \ldots$ as the recursion proceeds.) Similarly, a subformula containing ψ_1^- is introduced due to negative derived terms. The positive derived terms $u_1, u_2, \ldots, u_{m'}$ act as choice-points, because there may exist many clauses that unify with each u_i. Consider some choice of unifications for each u_i, then the recursion introduces more kernel constraints, non-monotonic acceptors, and negative subformulas. Let $\widehat{\beta}$ be an expansion of some β for particular unification choices of the positive derived terms appearing through the recursion. Then this expansion has the following form:

$$\widehat{\beta}(\varphi, M, \Theta) = \left(\bigwedge_i (\Theta_i \subseteq \mathbf{ker}\ \varphi)\right) \wedge \left(\bigwedge_i \alpha(P_i, N_i, \varphi, M)\right) \wedge \left(\bigwedge_i \begin{array}{l}\forall \varphi'\ \omega(\varphi, \varphi', V_i^+) \Rightarrow \\ \psi_i^-(\varphi', M, \Theta_i)\end{array}\right) \tag{21}$$

The following lemmas help to simplify the expansion.

Lemma 3. *Non-monotonic acceptors compose over conjunction for fixed φ and X.*

$$\alpha(P_a, N_a, \varphi, X) \wedge \alpha(P_b, N_b, \varphi, X) = \alpha(P_a \cup P_b, N_a \cup N_b, \varphi, X) \tag{22}$$

Lemma 4. *Constraints on the kernel of φ compose over conjunction.*

$$(\Theta_a \subseteq \mathbf{ker}\ \varphi) \wedge (\Theta_b \subseteq \mathbf{ker}\ \varphi) = (\Theta_a \oplus \Theta_b) \subseteq \mathbf{ker}\ \varphi \tag{23}$$

Applying the lemmas simplifies Equation 21 to:

$$\widehat{\beta}(\varphi, M, \Theta) = (\Theta' \subseteq \mathbf{ker}\ \varphi) \wedge \alpha(P', N', \varphi, M) \wedge \left(\bigwedge_i \begin{array}{l}\forall \varphi'\ \omega(\varphi, \varphi', V_i^+) \Rightarrow \\ \psi_i^-(\varphi', M, \Theta_i)\end{array}\right) \tag{24}$$

where

$$\Theta' = \Theta \oplus \left(\bigoplus \Theta_i\right) \tag{25}$$

$$P' = \bigcup P_i, \quad N' = \bigcup N_i \tag{26}$$

In summary, for a particular set of unification choices the backwards chaining reduces to:

1. Constraints on the kernel of φ, which equate variables,
2. A single non-monotonic acceptor containing only fact terms,
3. A number of backwards chaining predicates for negative derived terms.

A clause λ may have an exponential number of expansions $\widehat{\beta}$. Label these expansions $\widehat{\beta}_1, \ldots, \widehat{\beta}_{c_\lambda}$, then they relate to the original predicate through disjunction:

$$\beta_\lambda(\varphi, M, \Theta) = \bigvee_{1 \leq i \leq c_\lambda} \widehat{\beta}_i(\varphi, M, \Theta) \tag{27}$$

This decomposition also allows the ψ^- terms to be rewritten in terms of the expansion:

$$\psi^-(\varphi, M, \Theta) = \bigwedge_{mgu(w_i, h_{\lambda'}) \neq \emptyset} \bigwedge_{1 \leq j \leq c_{\lambda'}} \forall \varphi' \left[\begin{array}{l} \omega(\varphi, \varphi', V^+) \Rightarrow \\ \neg \widehat{\beta}_j(\varphi', M, \Theta \oplus mgu(w_i, h_{\lambda'})) \end{array} \right] \quad (28)$$

The decomposition of β_λ shows that each ψ^- term will expand into some number of non-monotonic acceptors depending on the depth of the negation, which is certainly finite. Unlike the positive derived terms, each ψ^- must examine all the relevant $\widehat{\beta}$ predicates to ensure that the negated derived term is not satisfied. Furthermore, the simplification lemmas cannot be directly applied to expansions of ψ^- because the non-monotonic acceptors appear in negated form. In the following sections we use these expansions to generate models from the backwards chaining proof trees.

4.2 Restratification

The previous analysis assumed that backwards chaining terminates at fact terms. This assumption can be violated if Λ contains clauses with fact terms as heads. The clause $task(x) \leftarrow taskmap(x, y)$ is an example. Fortunately, there is a simple syntactic operation that soundly manipulates Λ so that no fact terms appear as heads. We call this process *restratification*, because it changes the ordering $\prec$.

Definition 8. *Given Υ_F, Υ_D, Λ, and a goal G, then the* restratified *system is Υ_F, Υ_D^*, Λ^*, G^* where:*

1. *Introduce a new unary derived function symbol $restrat(\cdot)$ to Υ_R.*

$$\Upsilon_D^* = \Upsilon_D \cup \{restrat(\cdot)\} \quad (29)$$

2. *For each clause $\lambda \in \Lambda$ where the head h is a fact term, add the modified clause λ^* to Λ^*:*

$$restrat(h) \leftarrow \neg h, \quad s_1, s_2, \ldots, s_n, \quad \neg t_1, \neg t_2, \ldots, \neg t_m \quad (30)$$

3. *For each clause $\lambda \in \Lambda$ where the head h is derived term, add λ to Λ^**
4. *Modify the goal $G = (P_G, N_G)$ to include the negative derived term $\neg restrat(x)$ where x is a variable that does not appear in G.*

$$G^* = \Big(P_G, N_G \cup \{restrat(x)\} \Big) \quad (31)$$

Lemma 5. *If Λ is a set of non-recursive and stratified Horn clauses, then the restratified clauses Λ^* are also non-recursive and stratified where no clause has a fact term as head.*

Theorem 2. *The models that satisfy G are related to the models that satisfy G^* according to:*

$$\begin{aligned}&\forall M \subset \mathcal{T}_G(\Upsilon_F),\ \left(\Gamma_\Lambda(M) \cap \mathcal{T}_G(\Upsilon_F) = M\right) \Rightarrow \\ &\left(\exists \varphi\ \alpha(P_G, N_G, \varphi, \Gamma_\Lambda(M)) \Leftrightarrow \exists \varphi'\ \alpha(P_G^*, N_G^*, \varphi', \Gamma_{\Lambda^*}(M))\right)\end{aligned} \tag{32}$$

If M is a set of ground fact terms such that $\Gamma_\Lambda(M)$ does not grow the number of fact terms, then the restratified system will be in agreement with G. Conversely, when models are found that satisfy the restratified system, then these models do not grow fact terms under the original system. Of course, ground derived terms can still grow under either system. This is not a limitation, because solutions to G that are not solutions to G^* will accumulate fact terms under Γ_Λ until the set of ground fact terms is exactly a solution to G^*.

4.3 Generating Schedules: Part 1

We now apply these techniques to the running example of a scheduling abstraction. Our goal is to find a model that contains three tasks T_1, T_2, T_3 and two processors P_1, P_2 so that the tasks are scheduled onto the processors. Furthermore tasks T_1 and T_2 cannot be on the same processor; the same constraint holds for task T_2 and T_3. In order to make the example more interesting the tasks are not introduced in the goal:

$$\begin{aligned}P_G &= \{\, processor(p_1), processor(p_2), constraint(t_1, t_2), constraint(t_2, t_3)\,\} \\ P_N &= \{no\_map(x), bad\_map(y, z)\}\end{aligned} \tag{33}$$

for $\Upsilon_R = \{no\_map(\cdot), bad\_map(\cdot, \cdot)\}$ and all other function symbols in Υ_F.

Applying restratification to the original clauses yields:

$$restrat(task(x)) \leftarrow taskmap(x, y), \neg task(x) \tag{34}$$
$$restrat(processor(y)) \leftarrow taskmap(x, y), \neg processor(y) \tag{35}$$
$$restrat(task(x)) \leftarrow constraint(x, y), \neg task(x) \tag{36}$$
$$restrat(task(y)) \leftarrow constraint(x, y), \neg task(y) \tag{37}$$
$$no\_map(task(x)) \leftarrow task(x), \neg taskmap(x, y) \tag{38}$$
$$bad\_map(task(x), task(y)) \leftarrow taskmap(x, z), taskmap(y, z), constraint(x, y) \tag{39}$$

and the restratified goal is $G^* = (P_G, N_G \cup \{restrat(w)\})$.

Next, the backwards chaining predicates are expanded until the goal is expressed by a system of non-monotonic acceptors defined with only fact terms.

In this example the ω terms are trivial and have been removed.

$$
\begin{aligned}
&\exists M,\ \exists\varphi\ \beta_{G^*}(\varphi, M, \mathcal{ID}_{\mathcal{T}}) = \\
&\quad \exists M,\ \exists\varphi,\ \forall\varphi'\ \alpha(P_G, \emptyset, \varphi, M) \wedge \\
&\quad \neg\alpha\Big(\{task(x_1)\}, \{taskmap(x_1, y_1)\}, \varphi', M\Big) \wedge \\
&\quad \neg\alpha\Big(\{taskmap(x_2, y_2)\}, \{task(x_2)\}, \varphi', M\Big) \wedge \\
&\quad \neg\alpha\Big(\{taskmap(x_3, y_3)\}, \{processor(y_3)\}, \varphi', M\Big) \wedge \\
&\quad \neg\alpha\Big(\{constraint(x_4, y_4)\}, \{task(x_4)\}, \varphi', M\Big) \wedge \\
&\quad \neg\alpha\Big(\{constraint(x_5, y_5)\}, \{task(y_5)\}, \varphi', M\Big) \wedge \\
&\quad \neg\alpha\Big(\{taskmap(x_2, z_2), taskmap(y_2, z_2), constraint(x_2, y_2)\}, \emptyset, \varphi', M\Big)
\end{aligned}
\tag{40}
$$

After the goal has been reduced, the quantifiers must be eliminated from the formula. This elimination procedure is described in the next section.

5 Eliminating the Closed World

The formula $\exists M,\ \exists\varphi\ \beta_G(\varphi, M, \mathcal{ID}_{\mathcal{T}})$ contains a second-order variable M ranging over all the closed worlds, of which there are an infinite number. The next step in model generation is the elimination of the variable M. The *elimination* of M means that we construct a new formula $\exists\varphi\ \beta'_G(\varphi)$ that does not contain M, but the solutions to this formula can be used to construct a finite set of facts M satisfying the original formula.

Elimination is accomplished by constructing a finite candidate set M_C of non-ground terms with the property that there exists a satisfying M if and only if there exists a subset $M'_C \subseteq M_C$ where $\exists\varphi\ \beta_G(\varphi, \varphi(M'_C), \mathcal{ID}_{\mathcal{T}})$. Once M'_C is discovered, all that remains is to arbitrarily choose an assignment of variables to constants to get a concrete M.

In the interest of space, we describe the solution when all backwards chaining paths pass through at most one negated derived term. The results here are easily generalized to arbitrary depth of negation. (The previous example fits into this restricted case.) Consider any expansion $\widehat{\beta}_G$ of the goal predicate. The results from the previous section guarantee that it has the following simplified form:

$$
\begin{aligned}
&\exists M,\ \exists\varphi\ (\Theta' \subseteq \mathbf{ker}\ \varphi)\ \wedge\ \alpha(P', N', \varphi, M)\ \wedge \forall\varphi' \\
&\quad \Big[\neg\omega_1(\varphi, \varphi', V_1^+)\ \vee\ (\Theta'_1 \nsubseteq \mathbf{ker}\ \varphi')\ \vee \neg\alpha(P'_1, N'_1, \varphi', M)\Big] \wedge \\
&\quad \Big[\neg\omega_2(\varphi, \varphi', V_2^+)\ \vee\ (\Theta'_2 \nsubseteq \mathbf{ker}\ \varphi')\ \vee \neg\alpha(P'_2, N'_2, \varphi', M)\Big] \wedge \\
&\qquad\qquad \vdots \\
&\quad \Big[\neg\omega_k(\varphi, \varphi', V_k^+)\ \vee\ (\Theta'_k \nsubseteq \mathbf{ker}\ \varphi')\ \vee \neg\alpha(P'_k, N'_k, \varphi', M)\Big]
\end{aligned}
\tag{41}
$$

The simplification results in exactly one non-negated acceptor $\alpha(P', N', \varphi, M)$, where P' and N' are the composition of many non-monotonic acceptors according to Lemma 3. Each negated derived term also creates a backwards chaining tree according to Equation 28 and these trees can be simplified in the same way. After simplification, the negative derived terms yield disjunctions of negated ω formulas, kernel constraints, and non-monotonic acceptors. These negated subformulas are arbitrarily numbered $1, \ldots, k$ and primed to remind the reader that they result from simplification.

Recall that every clause has a least one positive term in the tail, so every P'_i must be non-empty. However, the goal G might not have positive terms, in which case $P' = \emptyset$. If this holds, let $M = \emptyset$, then M trivially satisfies $\alpha(\emptyset, N', \varphi, \emptyset)$ and trivial satisfies $\forall \varphi', \ \neg\alpha(P'_i, N'_i, \varphi', \emptyset)$. Thus, we focus on the interesting case where the goal G contains some positive terms, i.e. $P' \neq \emptyset$.

Assume $P' \neq \emptyset$, then by definition of α it must be that $\varphi(P') \subseteq M$. The set P' must have a homomorphic image in any solution M. Therefore, let the candidate solution $M_C = P'$. It may be that the negative part N' disallows some of these terms, but this can be discovered later. Next, consider the negated acceptors. It could be that some acceptor $\alpha(P'_i, N'_i, \varphi', M)$ is satisfied by the candidate model M_C. A necessary condition for this to occur is:

$$\exists \varphi' \ \omega_i(\varphi, \varphi', V_i^+) \wedge (\Theta'_i \subseteq \mathbf{ker}\ \varphi') \wedge (\varphi'(P'_i) \subseteq M_C) \tag{42}$$

This situation is only problematic if no term from N'_i has a homomorphic image in M_C. These problematic situations are mitigated by expanding M_C with the maximum number of negative terms: $M_C^{new} = M_C^{old} \cup \varphi'(N'_i)$. This expansion is performed for every possible φ', of which there are a finite number (if M_C is already finite). If N'_i contains variables not found in P'_i, then these variables are given new names.

Expanding M_C may provide new opportunities for negated acceptors to fail (i.e. the acceptor evaluates to *true*, so its negation is *false*). These new opportunities must be identified and may require further expansion of M_C. The strategy is to add the maximum number of terms to M_C that allow all the negated acceptors to succeed. For each negated acceptor assign an operator C_i^-:

$$C_i^-(\varphi, X) = \bigcup_{\varphi'} \left\{ \varphi'(N'_i) \;\middle|\; \omega_i(\varphi, \varphi', V_i^+) \wedge (\Theta'_i \subseteq \mathbf{ker}\ \varphi') \wedge (\varphi'(P'_i) \subseteq X) \right\} \cup X \tag{43}$$

For the single non-negated acceptor $\alpha(P', N', \varphi, M)$ assign the operator C^+:

$$C^+(\varphi, X) = X \cup \varphi(P') \tag{44}$$

The problem of finding the maximum M_C can now be stated in terms of a least fixed-point equation:

$$M_C = C^+(\varphi, M_C) = C_1^-(\varphi, M_C) = C_2^-(\varphi, M_C) = \ldots = C_k^-(\varphi, M_C) \tag{45}$$

These operators have two important properties: (1) *monotonic*: $X \subseteq C^{+/-}(X)$ (2) *extensive*: $X \subseteq Y \Rightarrow C^{+/-}(X) \subseteq C^{+/-}(Y)$. These properties lead to the important lemma:

Lemma 6. *If the least fixed-point M_C exists, then it is unique for a given φ.*

In fact, φ serves as a book-keeping mechanism to remember constraints over variables, and it can be constructed while solving the fixed-point equation. However, in the interest of space we omit the algorithm that constructs the fixed-point M_C.

Theorem 3. *For some expansion $\widehat{\beta}_G$, if the fixed-point M_C exists, then*

$$\forall M,\ \forall \varphi \quad \widehat{\beta}_G(\varphi, M, \mathcal{ID}) \Rightarrow \left(\exists M'_C \subseteq M_C,\ \exists \varphi' \quad \varphi'(M'_C) = M \right) \tag{46}$$

where φ' assigns variables to constants, and M is a minimal solution.

This key theorem explains that if a least fixed-point M_C exists for some expansion of the goal $\widehat{\beta}_G$, then a definite conclusion can be drawn about the satisfiability of this expansion: If there is a finite model that satisfies the subgoal, then there is some minimal model M that satisfies the subgoal. The minimal model M is exactly some subset $M'_C \subseteq M_C$ where the variables of M'_C have been assigned to constants. The least fixed-point M_C is finite, so there are a finite number of subsets M'_C. Furthermore, there are only a finite number of *interesting* ways that variables can be assigned constants. This result leads to an algorithm for model generation:

Model Generation Algorithm

```
1:  enumeration Results = { satisfiable, unsatisfiable, unknown };
2:  let result := unsatisfiable;
3:
4:  Restratify(Υ_F,Υ_D,Λ,G);
5:  let goal_expansions := {β̂_1, β̂_2, ..., β̂_k};
6:  for each β̂_i ∈ goal_expansions {
7:      if (fixed_point_exists(β̂_i) ) {
8:          let M_C = fixed_point(β̂_i);
9:          for each M'_C ⊆ M_C and each interesting φ' {
10:             if ( ∃φ β̂_i(φ, φ'(M'_C), ID)) {
11:                 result := satisfiable;
12:                 return φ'(M'_C);
13:         }
14:     }
15:     else result := unknown;
16: }
```

Fig. 3. Eliminating the closed world using fixed-points of goal expansions $\widehat{\beta}_i$.

5.1 Generating Schedules: Part 2

We apply these results to generate non-trivial models for the scheduling language. Recall that Equation 40 is the simplification of the goal predicate β_G and this predicate has only one expansion to $\widehat{\beta}$. The task is the calculation of the fixed-point M_C from $\widehat{\beta}$. Initially M_C contains only the positive part of the goal:

$$M_C^0 = \{processor(p_1), processor(p_2), constraint(t_1, t_2), constraint(t_2, t_3)\} \quad (47)$$

This candidate model may violate the negated acceptors:

$$\forall \varphi' \ \neg\alpha(\{constraint(x_4, y_4)\}, \{task(x_4)\}, \varphi', M) \wedge \\ \forall \varphi' \ \neg\alpha(\{constraint(x_5, y_5)\}, \{task(y_5)\}, \varphi', M)$$

The substitution $\varphi'(x_4) = \varphi'(x_5) \mapsto t_1,\ \varphi'(y_4) = \varphi'(y_5) \mapsto t_2$ is a witness to this possibility. (There exists a similar substitution for t_2, t_3.) These substitutions expand the candidate set to include the three tasks:

$$M_C^1 = M_C^0 \cup \{task(t_1), task(t_2), task(t_3)\} \quad (48)$$

This candidate set does not contain *taskmap* terms from tasks to processors, violating the subformula:

$$\forall \varphi' \ \neg\alpha(\{task(x_1)\}, \{taskmap(x_1, y_1)\}, \varphi', M)$$

The expansion of M_C introduces the *taskmap* terms and also new variables:

$$M_C^2 = M_C^1 \cup \{taskmap(t_1, x), taskmap(t_2, y), taskmap(t_3, z)\} \quad (49)$$

Finally, new processor terms are introduced for the end-points of the *taskmap* terms:

$$M_C^3 = M_C^2 \cup \{processor(x), processor(y), processor(z)\} \quad (50)$$

This set is the fixed-point, i.e. $M_C = M_C^3$.

The model generation problem can be solved by examining subsets of the fixed-point M_C. Some subsets will not produce satisfying models:

$$M_{fail} = \left\{ \begin{array}{c} task(t_1), task(t_2), task(t_3), \\ taskmap(t_1, x), taskmap(t_2, y), taskmap(t_3, z) \\ constraint(t_1, t_2), constraint(t_2, t_3), processor(p_1), \end{array} \right\} \quad (51)$$

where $x = y = z = p_1$. This subset will fail for any assignment of variables to constants because there are not enough distinct processors. On the other hand, the set

$$M_{success} = M_{fail} \cup \left\{ processor(p_2) \right\} \quad (52)$$

where $x = z = p_1$ and $y = p_2$ and $x \neq y$ satisfies the goal. In fact, any choice of constants that respects the equalities/disequalities satisfies the goal. In the next section we show how these subsets can be calculated using boolean satisfiability.

5.2 A Better Algorithm Using SMT

The simple algorithm in Figure 3 is a brute force approach for finding a model that satisfies the goal. It tries every subset of M_C and every interesting assignment φ' of variables to constants. In general, there are an exponential number of (M'_C, φ') pairs to test. This exponential blow-up cannot be eliminated entirely, but it can be mitigated by translating the problem into a SAT problem. Mature SAT algorithms can be used to suppress the exponential blow-up. The encoding described here assumes a modern solver capable of reasoning about equalities among a set of non-boolean variables. We use an SMT solver (*Satisfiability Modulo Theories*) called *Z3* [17] to accomplish this task. Z3 utilizes efficient SAT algorithms to solve problems that are not purely boolean; e.g. problems with equalities over non-boolean variables.

The first step of the encoding is the translation of complex terms to boolean variables. At this stage all the terms that need to be considered already exist in M_C, so the encoding is simple. Assign a boolean variable τ_i to each term t_i in the candidate set M_C. If τ_i is true then the corresponding term t_i is in the solution, otherwise t_i is not in the solution. Furthermore, introduce a set of non-boolean variables X, which are the variables occurring as subterms in M_C: $X = vars(M_C)$. We provide these non-boolean variables so the SAT solver can decide if some variables x_i, x_j in M_C should take the the same values ($x_i = x_j$), or different values ($x_i \neq x_j$), or a fixed value ($x_i = \mathsf{c}$).

The non-negated acceptor $\alpha(P', N', \varphi, M)$ defines terms P' that must be in any solution. Let $t_1^+, t_2^+, \ldots, t_k^+$ be the terms in M_C that were added according to P', then the following boolean formula must be true:

$$\bigwedge_{1 \leq j \leq k} \tau_j^+ \tag{53}$$

Next, consider any negated acceptor $\neg\alpha(P'_i, N'_i, \varphi', M)$. Let φ' be such that $\omega_i(\varphi, \varphi', V_i^+)$ holds, $\Theta'_i \subseteq \mathbf{ker}\ \varphi'$, and $\varphi'(P'_i) \subseteq M_C$. Let $\varphi'(P'_i) = \{t_1^-, t_2^-, \ldots, t_k^-\}$. (Note φ' may not exist.) The negated acceptor is satisfied if one of the following holds:

1. One of the t_j^- terms is not in the solution.
2. There exists a pair of variables $(x, y) \in vars(\varphi'(P'_i))$ that is also in the kernel of φ', but $x \neq y$.
3. There exists a variable $x \in vars(\varphi'(P'_i))$ and a constant c where $(x, \mathsf{c}) \in \mathbf{ker}\ \varphi'$, but $x \neq \mathsf{c}$.
4. There exists an extension of φ' to φ'' and some $t^+ \in M_C$ so that $t^+ \in \varphi''(N'_i)$.

Conditions (1)-(3) yield the following encoding:

$$\left(\bigvee_{1 \leq j \leq k} \neg\tau_j^-\right) \vee \left(\bigvee_{\substack{x, y \in vars(\varphi'(P'_i)),\\ (x,y) \in \mathbf{ker}\ \varphi'}} x \neq y\right) \vee \left(\bigvee_{\substack{x \in vars(\varphi'(P'_i)), \mathsf{c} \in \Sigma,\\ (x,\mathsf{c}) \in \mathbf{ker}\ \varphi'}} x \neq \mathsf{c}\right) \tag{54}$$

Condition (4) has a more complicated encoding:

$$\bigvee_{t^+}\left[\tau^+ \wedge \left(\bigwedge_{\substack{x,y \in vars(\varphi''(P_i') \cup t^+)),\\ (x,y) \in \mathbf{ker}\ \varphi''}} x = y\right) \wedge \left(\bigwedge_{\substack{x \in vars(\varphi''(P_i') \cup t^+)),\\ \mathsf{c} \in \Sigma, (x,\mathsf{c}) \in \mathbf{ker}\ \varphi''}} x = \mathsf{c}\right)\right] \tag{55}$$

Such an encoding must be generated for all relevant φ' and φ''. A similar translation encodes the negative part N' of the non-negated acceptor α'. We omit it this in the interest of space.

In this discussion we have ignored the relationship between variables and terms. For example, two terms $f(x)$, $f(y)$ are affected by equating (or disequating) the variables: If $x = y$ ($x \neq y$) then $f(x) = f(y)$ ($f(x) \neq f(y)$). In this case the relationship can be easily encoded because all the variables in M_C will only take constant values. Let t_i, t_j be two terms that unify by equating variables to other variables or constants. Then the following must hold:

$$(\tau_i \Leftrightarrow \tau_j) \vee \left(\bigvee_{(x,y) \in mgu(t_i,t_j)} x \neq y\right) \vee \left(\bigvee_{(x,\mathsf{c}) \in mgu(t_i,t_j)} x \neq \mathsf{c}\right) \tag{56}$$

Terms that unify by assigning some variables to complex terms are ignored.

6 Conclusion and Future Work

Model generation is an important tool for the model-based design of software systems. It can be used to generate non-trivial solution instances from domain-specific abstractions, perform design-space exploration, and reason about model-transformations. We have given a sound algorithm that generates models from non-recursive and stratified Horn logic. These algorithms have been implemented in a tool called *FORMULA* (FORmal Modeling Using Logic Analysis). The SMT (SAT Modulo Theories) solver Z3 is used to solve the SAT encodings output by FORMULA.

Future work includes extending model generation to encompass *constraint logic programming* (CLP) frameworks. CLP combines Horn logic with constraints, as in the following clause:

$$bad\_sched(critial\_task(x)) \leftarrow critical\_task(x), task(y), priority(x) < priority(y)$$

Assume there is a new type of task called a *critical_task*. This clause states that a bad schedule assigns a high priority to a non-critical task. Priorities are expressed by the ordering $<$ over integers, resulting in a combination of Horn logic with the theory of integers. We will utilize additional theories available at the SMT level to generate models for CLP extensions.

7 Acknowledgments

We would like to thank Nikolaj Bjørner for his invaluable feedback and his insight into Z3.

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: Mda guide version 1.0.1. Technical report (2003)
3. Bezivin, J., Gerbé, O.: Towards a precise definition of the omg/mda framework. In Proceedings of the 16th Conference on Automated Software Engineering (November 2001) 273–280
4. G. Karsai, J. Sztipanovits, A.L.T.B.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (January 2003) 145–164
5. J. Burch, R. Passerone, A.S.V.: Modeling techniques in design-by-refinement methodologies. In: Integrated Design and Process Technology. (June 2002)
6. Lee, E.A., Neuendorffer, S.: Actor-oriented models for codesign: Balancing re-use and performance. Formal Methods and Models for Systems, Kluwer (2004)
7. Przymusinski, T.C.: Every logic program has a natural stratification and an iterated least fixed point model. In: PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (1989) 11–21
8. Jensen, T.R., Toft, B.: Graph Coloring Problems. Wiley-Interscience, New York, ISBN 0-471-02865-7
9. Reiter, R.: On closed world data bases. In: Readings in nonmonotonic reasoning, Morgan Kaufmann Publishers Inc. (1987) 300–310
10. Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Computation theory and logic, Springer-Verlag (1987) 20–36
11. van Gelder, A., Ross, K., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K., eds.: Proceedings of the Fifth International Conference on Logic Programming, Cambridge, Massachusetts, The MIT Press (1988) 1070–1080
13. Marek, V.W., Nerode, A., Remmel, J.B.: A context for belief revision: Forward chaining - normal nonmonotonic rule systems. Ann. Pure Appl. Logic **67**(1-3) (1994) 269–323
14. Emden, M.H.V., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4) (1976) 733–742
15. Apt, K.R., Doets, K.: A new definition of SLDNF-resolution. The Journal of Logic Programming **18**(2) (February 1994) 177–190
16. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3) (2001) 374–425
17. de Moura, L., Bjørner, N.: 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)