Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 213 (2008) 75–94
www.elsevier.com/locate/entcs

An Algorithm for Approximating the Satisfiability Problem of High-level Conditions

Karl-Heinz Pennemann[1,2]

Department of Computing Science
University of Oldenburg
D-26111 Oldenburg, Germany

Abstract

The satisfiability problem is the fundamental problem in proving the conflict-freeness of specifications, or in finding a counterexample for an invalid statement. In this paper, we present a non-deterministic, monotone algorithm for this undecidable problem on graphical conditions that is both correct and complete, but in general not guaranteed to terminate. For a fragment of high-level conditions, the algorithm terminates, hence it is able to decide. Instead of enumerating all possible objects of a category to approach the problem, the algorithm uses the input condition in a constructive way to progress towards a solution. To this aim, programs over transformation rules with external interfaces are considered. We use the framework of weak adhesive HLR categories. Consequently, the algorithm is applicable to a number of replacement capable structures, such as Petri-Nets, graphs or hypergraphs.

Keywords: first-order satisfiability problem, high-level conditions, high-level programs, graph transformation, weak adhesive HLR categories.

[1] This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 (Graduate School on Trustworthy Software Systems) and HA 2936/2 (Development of Correct Graph Transformation Systems). Thanks to Arend Rensink for the visit to the University of Twente, during which the preliminary ideas for this work were discussed, to Annegret Habel for constructive remarks concerning this paper, and to the referees for their thorough reviews.

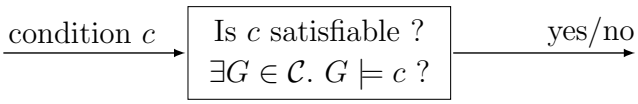
[2] Email: Pennemann@Informatik.Uni-Oldenburg.de

doi:10.1016/j.entcs.2008.04.075

1 Introduction

(High-level) Conditions are a graphical formalism to specify valid objects as well as morphisms, i.e., they can be used to describe system or program states as well as specify matches for transformation rules. They provide an intuitive formalism for structural properties and are well suited for reasoning about the behavior of transformation systems.

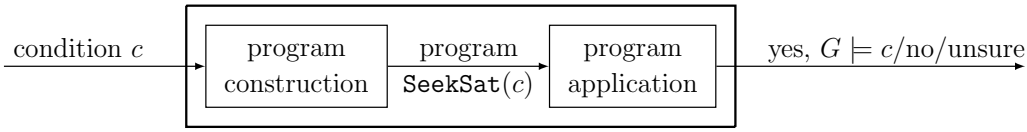
For a given category $\mathcal{C}$ of objects, the satisfiability problem is the problem to decide for any given condition c, whether or not $\exists G \in \mathcal{C}.\ G \models c$.



The satisfiability problem can be used to show that a specification is conflict free or to prove that a statement is invalid, i.e. if the negated statement is satisfiable. If some object G is provided along with a positive answer, one yields a counterexample for the latter case, illustrating an invalid system state. In this sense, a satisfiability algorithm complements a first-order theorem prover, with the prover searching for proof and the satisfiability algorithm looking for a counterexample. For the category **Graph** of finite, directed, labeled graphs, conditions are expressively equivalent to first order logic on graphs [20,11]. Therefore the satisfiability problem for arbitrary conditions over arbitrary categories is not decidable, i.e., there does not exist an algorithm that decides the satisfiability of arbitrary conditions over arbitrary categories. Still, an approximation of this undecidable problem is possible, but necessarily either unsound, incomplete or not guaranteed to terminate.

In this paper, we present a sound and complete algorithm that works for conditions over a class of replacement capable categories. Instead of enumerating all possible objects of a category to approach the problem, the presented algorithm uses the input condition in a constructive way. Starting from the initial object, e.g. the empty graph, elements of positive statements are added if necessary, while the absence of forbidden patterns is checked. The result is a monotone (non-deleting) algorithm which non-deterministically progresses towards a satisfiable object. Technically, we generate for each condition a program `SeekSat`. As we need to handover information between computation steps, `SeekSat` works on morphisms of the considered category. To this aim,

programs over transformation rules with external interfaces are considered.



The paper is organized as follows. In Section 2, the definition of conditions is reviewed and programs over rules with external interfaces for high-level structures such as graphs are introduced. In Section 3, the satisfiability algorithm is presented, its correctness and completeness is shown, and for a fragment of conditions, its termination and hence its capability to decide is proved. In Section 4, practical aspects concerning a possible implementation and optimization are discussed. We relate our results to other work in Section 5. A conclusion including further work is given in Section 6. A long version of this paper is available at [1] including detailed proofs.

2 Conditions and Rules

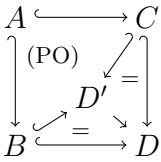
In this section, we review the definitions of conditions and introduce programs over rules with external interfaces for high-level structures such as graphs. We seek an algorithm for the satisfiability problem of conditions that is not concerned with a specific definition of a structure. Therefore, we use the framework of weak adhesive HLR categories introduced as combination of HLR systems and adhesive categories. A detailed introduction can be found in [9,8].

For a given category $\mathcal{C}$, let Mor be the set of all morphisms.

Assumption. Assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category [8] consisting of a category $\mathcal{C}$ of objects and a class $\mathcal{M} \subseteq \text{Mor}$ of monomorphisms. Additionally, we require

- a $\mathcal{M}$-*initial* object I, i.e., an object $I \in \mathcal{C}$ such that for every object $G \in \mathcal{C}$ there is a unique morphism $i_G \colon I \to G$ in $\mathcal{M}$, called the *initial morphism* to G,
- *epi-$\mathcal{M}$-factorization*, i.e., for every morphism there is an epi-mono-factorization with monomorphism in $\mathcal{M}$,
- a *finite length of $\mathcal{M}$-decompositions*, i.e., for every morphism m in $\mathcal{M}$, the length of every decomposition $m_n \circ \ldots \circ m_1 = m$ consisting of non-epimorphisms m_j in $\mathcal{M}$ $(1 \leq j \leq n)$ is finite,
- a *finite number of $\mathcal{M}$-matches*, i.e. for every morphism $l \colon K \hookrightarrow L$ in $\mathcal{M}$ and every object G, there exist only a finite number of morphisms $m \colon L \hookrightarrow G$ in $\mathcal{M}$ s.t. $\langle l, m \rangle$ has a pushout complement, and

- the *pullback-pushout-$\mathcal{M}$ property*, i.e., for every pair of $\mathcal{M}$-morphisms $B \hookrightarrow D \hookleftarrow C$, the unique morphism $D' \to D$ of the pushout $\langle B \hookrightarrow D' \hookleftarrow B\rangle$ of the pullback $\langle B \hookleftarrow A \hookrightarrow C\rangle$ of $B \hookrightarrow D \hookleftarrow C$ is in $\mathcal{M}$.

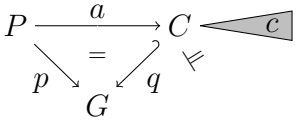


Notation. A morphism m with domain A and codomain B is denoted by $m\colon A \to B$. $m\colon A \hookrightarrow B$ indicates that m is a morphism in $\mathcal{M}$. For an object G, let $\mathrm{id}_G\colon G \leftrightarrow G$ be the *identity* on G and let $i_G\colon I \to G$ be the initial morphism to G. For a morphism, the actual mapping is conveyed by indices, if necessary.

Example 2.1 The category **Graph** of finite, directed, labeled graphs [8] together with the class $\mathcal{M}$ of all injective graph morphisms constitutes a weak adhesive HLR category that satisfies the assumptions. The empty graph $\emptyset$ is the $\mathcal{M}$-initial object.

Conditions are nested constraints and application conditions generalizing the corresponding notions in [14,7] along the lines of [20].

Definition 2.2 (conditions) A *(nested) condition* over an object P is of the form $\exists a$ or $\exists(a, c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions over P yield conditions over P, i.e., true, $\neg c$ and $\wedge_{j\in J} c_j$ are (Boolean) conditions over P, where J is a finite index set and c, $(c_j)_{j\in J}$ are conditions over P. Additionally, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, false abbreviates $\neg$true, $\vee_{j\in J} c_j$ abbreviates $\neg\wedge_{j\in J} \neg c_j$ and $c \Rightarrow d$ abbreviates $\neg c \vee d$.



An object G *satisfies* a condition $\exists a$ $[\exists(a, c)]$, if the condition is over the initial object I and the initial morphism $i_G\colon I \to G$ satisfies the condition. A morphism p *satisfies* a condition $\exists a$ $[\exists(a, c)]$ if there exists a morphism q in $\mathcal{M}$ such that $q \circ a = p$ [and q satisfies c]. The satisfaction of conditions by objects [by morphisms] is extended onto Boolean conditions in the usual way. We write $G \models c$ $[p \models c]$ to denote that object G [morphism p] satisfies c.

In the context of objects, conditions (over the initial object I) are also called *constraints*. In the context of rules, conditions are also called *application*

conditions.

Notation. For a morphism $a\colon P \to C$ in a condition, we just depict the codomain C, if the domain P can be unambiguously inferred, i.e. for application conditions over some left-hand side L of a rule and for constraints. For instance, the constraint $\forall(\emptyset \to \bigcirc_1, \exists(\bigcirc_1 \to \bigcirc_1\!\!\rightarrow\!\!\bigcirc_2))$ with the meaning "Every node has an outgoing edge to another distinct node" can be represented by $\forall(\bigcirc_1, \exists(\bigcirc_1\!\!\rightarrow\!\!\bigcirc_2))$.

A condition is in $\mathcal{M}$-*normal form* ($\mathcal{M}$NF), if for every subcondition $\exists a$ and $\exists(a, c)$ the morphism a is in $\mathcal{M}$.

Fact 2.3 ($\mathcal{M}$-normal form) *Every condition c over P can be transformed into a condition c' in $\mathcal{M}$NF such that, for all morphism $p\colon P \hookrightarrow G$ in $\mathcal{M}$, $p \models c$ if and only if $p \models c'$.*

Proof Substitute every subcondition $\exists a$ and $\exists(a, c)$, $a \notin \mathcal{M}$, with false (see [11]). □

For the definition of rules with external interfaces, we define partial morphisms.

Definition 2.4 (partial morphisms) For a given category $\mathcal{C}$, a *partial morphism* from A to B, denoted by $A \rightharpoonup B$, is a span of morphisms $\langle A \hookleftarrow K \to B\rangle$, consisting of morphisms $K \hookrightarrow A$ in $\mathcal{M}$ and $K \to B$ in Mor. Two partial morphism $a, b\colon A \rightharpoonup B$ are commutative, denoted by $a = b$, if there is an isomorphism $K_a \leftrightarrow K_b$ such that the resulting triangles commute. The set of all *partial morphisms* is denoted by PMor.

Fact 2.5 (closure under composition) *Partial morphism are closed under composition, i.e., $A \rightharpoonup B$ and $B \rightharpoonup C$ can be composed to $A \rightharpoonup C$.*

Proof By pullback construction. The result is unique up to ismorphism. □

Fact 2.6 *Every morphism is also a partial morphism: $\mathcal{M} \subseteq$ Mor $\subseteq$ PMor.*

We require rule applications restricted to a certain context. Therefore, we consider rules with external interface and declare transformations of morphisms instead of objects. In this paper, we consider only matches in $\mathcal{M}$.

Definition 2.7 (rules with external interface) A *rule* $\rho = \langle\langle X \rightharpoonup L \hookleftarrow K \hookrightarrow R\rangle, \mathrm{ac}_L\rangle$ consists of a partial morphism $x\colon X \rightharpoonup L$, the *external interface*, two $\mathcal{M}$-morphisms $l\colon K \hookrightarrow L$, $r\colon K \hookrightarrow R$, and a *(left) application condition*

ac_L over L.

$$\begin{array}{ccccccc} X & \xrightarrow{x} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ & \searrow^{m'} & \downarrow{\scriptstyle m} & (1) & \downarrow{\scriptstyle d} & (2) & \downarrow{\scriptstyle m^*} \\ & & G & \xleftarrow[l^*]{} & D & \xrightarrow[r^*]{} & H \end{array}$$

A direct derivation from a (partial) morphism m' to a morphism m^*, denoted by $m' \Rightarrow_{\rho,m} m^*$, is defined by a $\mathcal{M}$-morphism $d\colon K \hookrightarrow D$ and two pushouts $\langle m, l^* \rangle$ and $\langle r^*, m^* \rangle$, if $m' = m \circ x$ and $m \models \mathrm{ac}_L$. We will often refer to the morphisms m' and m^* as the *input* and the *result* of the derivation, respectively. As the *match* $m\colon L \to G$ is in $\mathcal{M}$, we speak of $\mathcal{M}$*-matching*.

External interfaces may be seen as a kind of input/output types. They can be used to control the location of rule applications. Rules with external interface object I correspond to usual transformation rules. For now, to concatenate rule applications, an external interface object X has to coincide with the right-hand side of the predecessing rule. The external interface may be a partial morphism to selectively use the interface information it provides, see Example 2.9. The input may be a partial morphism, if programs over rules with external interfaces are considered.

Remark 2.8 Intentionally, we only consider an interface on the left-hand side, as a rule author should just have to care about the "input" interface. In case of external interfaces for left- and right-hand side, it suffices to consider total interface morphisms. However, one may have to write a set of similar rules, depending on rules that follow in sequential composition.

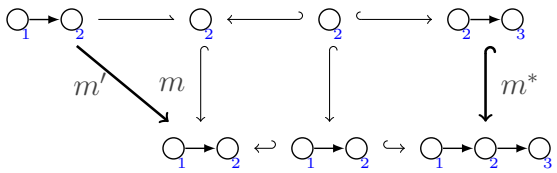
Notation. As every span $\langle L \hookleftarrow K \hookrightarrow R \rangle$ of morphisms in $\mathcal{M}$ can be seen as a partial morphism, we write $\langle \langle X \rightharpoonup L \Rightarrow R \rangle, \mathrm{ac}_L \rangle$. If $\mathrm{ac}_L = \mathrm{true}$, we omit the left application condition and write $\langle X \rightharpoonup L \Rightarrow R \rangle$. In case the external interface is the initial object, i.e. $X = I$, we just write $\langle L \Rightarrow R \rangle$.

Example 2.9 (rule with external interface) Consider the graph rule

$$\rho = \langle \bigcirc_1 \!\!\to\!\! \bigcirc_2 \rightharpoonup \bigcirc_2 \hookleftarrow \bigcirc_2 \hookrightarrow \bigcirc_2 \!\!\to\!\! \bigcirc_3 \rangle$$

that, for a graph morphism $\bigcirc_1 \!\!\to\!\! \bigcirc_2 \to G$, adds an edge from the image of node 2 to a newly created node. With an external interface, it becomes possible to hand over information between derivation steps without the use of additional elements. In this example, the external interface expresses that a given chain of nodes in a graph, represented by the last two nodes, must be extended. We will make use of similar handover effects in our satisfiability algorithm. As $X = R$, this rule is iterable. Note, the indices do not correspond with the

identities of the nodes, but convey their mappings.



For modeling transactions, we consider programs [13,19,12] on transformation rules with external interface.

Definition 2.10 (programs) *(High-level) Programs* are inductively defined: `Abort`, `Skip` and every rule ρ with external interface are programs. For programs $\mathtt{P}, \mathtt{Q}$ and a condition c, every finite set $\mathcal{S}$ of programs, $\mathtt{Fix(P)}$, $(\mathtt{P};\mathtt{Q})$, $\mathtt{if}\, c\, \mathtt{then}\, \mathtt{P}\, \mathtt{fi}$, $\mathtt{P}^*$, $\mathtt{P}{\downarrow}$ and $\mathtt{while}\, c\, \mathtt{do}\, \mathtt{P}\, \mathtt{od}$ are programs.

While $\mathtt{Fix(P)}$ is an *interface manipulation*, a program $\mathcal{S}$ denotes the *(demonic) nondeterministic choice*, $(\mathtt{P};\mathtt{Q})$ is the *sequential composition*, $\mathtt{if}\, c\, \mathtt{then}\, \mathtt{P}\, \mathtt{fi}$ is the *conditional execution*, $\mathtt{P}^*$ is the *reflexive, transitive closure*, $\mathtt{P}{\downarrow}$ is the *as long as possible iteration*, and $\mathtt{while}\, c\, \mathtt{do}\, \mathtt{P}\, \mathtt{od}$ is the *conditional iteration* of programs.

To reflect the presence of rules with external interface, the semantics of a program $\mathtt{P}$ is a ternary relation on partial morphisms, denoted by $[\![\mathtt{P}]\!] \subseteq \mathrm{PMor}^3$, instead of a binary relation on objects. The first two morphisms represents input and result while the last morphism is an "interface relation" from the domain of the input to the domain of the result morphism.

$$
\begin{aligned}
[\![\mathtt{Abort}]\!] &= \emptyset \\
[\![\mathtt{Skip}]\!] &= \{\langle m', m', \mathrm{id}\rangle \mid m' \in \mathrm{PMor}\} \\
[\![\rho]\!] &= \{\langle m', m^*, m_\rho\rangle \mid m' \Rightarrow_{\rho,m} m^*\} \\
[\![\mathtt{Fix(P)}]\!] &= \{\langle m', m^* \circ m_\mathtt{P}, \mathrm{id}\rangle \mid \langle m', m^*, m_\mathtt{P}\rangle \in [\![\mathtt{P}]\!]\} \\
[\![(\mathtt{P};\mathtt{Q})]\!] &= \{\langle m', m^*, m_\mathtt{Q} \circ m_\mathtt{P}\rangle \mid \langle m', m, m_\mathtt{P}\rangle \in [\![\mathtt{P}]\!], \langle m, m^*, m_\mathtt{Q}\rangle \in [\![\mathtt{Q}]\!]\} \\
[\![\mathcal{S}]\!] &= \textstyle\bigcup_{\mathtt{P}\in\mathcal{S}} [\![\mathtt{P}]\!] \\
[\![\mathtt{if}\, c\, \mathtt{then}\, \mathtt{P}\, \mathtt{fi}]\!] &= \{\langle m', m^*, m_\mathtt{P}\rangle \in [\![\mathtt{P}]\!] \mid m' \models c\} \cup \{\langle m', m', \mathrm{id}\rangle \mid m' \models \neg c\} \\
[\![\mathtt{P}^*]\!] &= [\![\{\mathtt{Skip}, (\mathtt{Fix(P)};\mathtt{P}^*)\}]\!] \\
[\![\mathtt{P}{\downarrow}]\!] &= \{\langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{P}^*]\!] \mid \nexists m.\langle m^*, m, \mathrm{id}\rangle \in [\![\mathtt{Fix(P)}]\!]\} \\
[\![\mathtt{while}\, c\, \mathtt{do}\, \mathtt{P}\, \mathtt{od}]\!] &= \{\langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{if}\, c\, \mathtt{then}\, \mathtt{P}\, \mathtt{fi}^*]\!] \mid m^* \models \neg c\}
\end{aligned}
$$

where $m_\rho\colon X \rightharpoonup R$ is the partial *ρ-induced morphism* $X \rightharpoonup L \hookleftarrow K \hookrightarrow R$ and id is the identity on the domain of m' (and m^*).

Remark 2.11 Programs of the form $(\mathtt{P};(\mathtt{Q};\mathtt{R}))$ and $((\mathtt{P};\mathtt{Q});\mathtt{R})$ can be proved to be equal; by convention, both can be written as $\mathtt{P};\mathtt{Q};\mathtt{R}$.

Fact 2.12 *Every program* P *in the sense of [13,12] can be seen as a program over rules with the initial object as external interface:* $\langle G, H\rangle \in [\![\mathtt{P}]\!]$ *in the sense of [12] if and only if* $\langle i_G, i_H, \mathrm{id}_I\rangle \in [\![\mathtt{P}']\!]$*, where* P′ *is yielded from* P *by substituting every elementary program* $\langle L \Rightarrow R\rangle$ *with* $\mathtt{Fix}(\langle I \to L \Rightarrow R\rangle)$.

Example 2.13 (Fix operator) The Fix operation is not a true computation step, but an interface manipulation, mainly used for normalization and in context of sequential composition. For the rule AddNode : $\langle \emptyset \to \emptyset \Rightarrow \bigcirc\rangle$, the use of Fix is illustrated in Figure 1: We will find $\langle m', m^*, m_{\mathtt{AddNode}}\rangle \in [\![\mathtt{AddNode}]\!]$ if and only if $\langle m', m^* \circ m_{\mathtt{AddNode}}, \mathrm{id}_\emptyset\rangle \in [\![\mathtt{Fix}(\mathtt{AddNode})]\!]$. The different semantics are emphasized by bold morphisms.

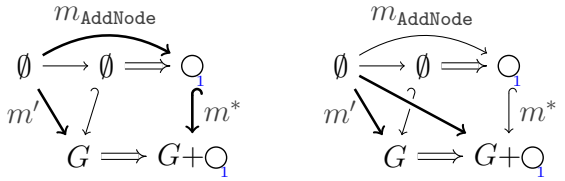


Figure 1. AddNode and Fix(AddNode)

Example 2.14 (sequential composition) We distinguish two uses of the sequential composition: with and without handover. Consider the rules

$$\begin{aligned}
&\mathtt{AddNode}: && \langle\, \emptyset \to \emptyset \Rightarrow \bigcirc \,\rangle\\
&\mathtt{DeleteNodeN}: && \langle\, \bigcirc \to \bigcirc \Rightarrow \emptyset \,\rangle\\
&\mathtt{DeleteNode}: && \langle\, \emptyset \to \bigcirc \Rightarrow \emptyset \,\rangle
\end{aligned}$$

The program AddNode; DeleteNodeN has no observable effect: a node is added in the first step and deleted in the second step (sequential composition with handover). The program Fix(AddNode); DeleteNode also adds a node in the first step; in the second step however, a nondeterministically chosen, isolated node is deleted (sequential composition without handover). The bold morphisms in Figure 2 mark the differences in the external interface condition of the second rule application.

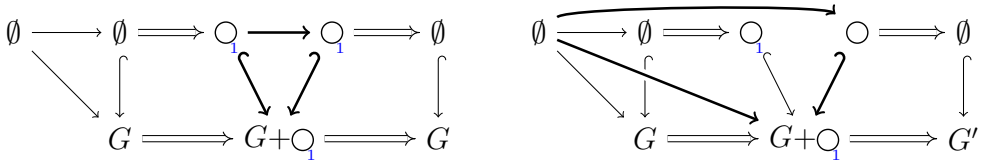


Figure 2. Sequential composition with and without handover

In the following, we will consider only non-deleting rules with external interface in $\mathcal{M}$. For a morphism $a\colon A \to B$, let $\mathrm{dom}(m) = A$ and $\mathrm{codom}(m) = B$.

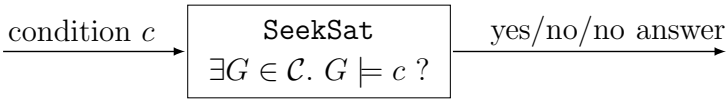
Fact 2.15 (every morphism in $\mathcal{M}$) *For non-deleting rules with external interface in $\mathcal{M}$, i.e. rules of the form $\rho = \langle X \hookrightarrow L \hookleftarrow K \hookrightarrow R \rangle$, the ρ-induced morphism $m_\rho \colon X \hookrightarrow R$ is in $\mathcal{M}$. For every derivation $m' \Rightarrow_{\rho,m} m^*$, the morphism $m^* \circ m_\rho$ is in $\mathcal{M}$ and there is a morphism $m^*_\rho \colon \mathrm{codom}(m') \hookrightarrow \mathrm{codom}(m^*)$ in $\mathcal{M}$ such that $m^*_\rho \circ m' = m^* \circ m_\rho$. Consequently, for every program computation $\langle m', m^*, m_{\mathtt{P}} \rangle \in [\![\mathtt{P}]\!]$, the morphism $m_{\mathtt{P}} \colon \mathrm{dom}(m') \hookrightarrow \mathrm{dom}(m^*)$ is in $\mathcal{M}$ and there is a morphism $m^*_{\mathtt{P}} \colon \mathrm{codom}(m') \to \mathrm{codom}(m^*)$ with $m^*_{\mathtt{P}} \circ m' = m^* \circ m_{\mathtt{P}}$. Furthermore m' in $\mathcal{M}$ implies m^* in $\mathcal{M}$.*

3 The Satisfiability Problem

After we give a formal definition of the satisfiability problem of high-level conditions, we will present a construction that, for any given condition, will yield a program over rules with external interface, trying to construct a satisfiable object.

Definition 3.1 (satisfiability problem) For a given category $\mathcal{C}$, the *satisfiability problem* is the problem to decide for any given condition c, whether or not $\exists G \in \mathcal{C}.\ G \models c$.

Like the satisfiability problem of first-order logic on graphs and on finite structures in general [21,5], the satisfiability problem of graph conditions and high-level conditions in general is undecidable [11]. We seek a correct and complete algorithm, not always guaranteed to terminate. The algorithm answers yes, as soon a result is found, answers no, if it terminates without results, and does not answer in case of non-termination.



The idea of our algorithm is to use the given condition in a constructive way by adding elements of positive statements if necessary while checking the absence of forbidden elements. The result is a monotone algorithm which non-deterministically progresses towards an object satisfying the input condition.

Theorem 3.2 (SeekSat) *For each condition c, there is a program $\mathtt{SeekSat}(c)$ that is correct and complete, i.e.,*

$$\langle \mathrm{id}_I, i_M, \mathrm{id}_I \rangle \in [\![\mathtt{SeekSat}(c)]\!] \quad \textit{implies} \quad M \models c,$$

$$(\exists H \in \mathcal{C}.\ H \models c) \quad \textit{implies} \quad \exists M \in \mathcal{C}.\ \langle \mathrm{id}_I, i_M, \mathrm{id}_I \rangle \in [\![\mathtt{SeekSat}(c)]\!].$$

Satisfaction of conditions by objects is defined by presence (or absence) of morphisms. For each condition c, we define two programs $\mathtt{Sat}(c)$ and $\overline{\mathtt{Sat}}(c)$, that for a given input $p'\colon P \hookrightarrow G$ in $\mathcal{M}$ are supposed to deliver some results $p^*\colon P \hookrightarrow H$ in $\mathcal{M}$, with the properties $p^* \models c$ and $p^* \not\models c$, respectively.

Construction. (SeekSat) For a condition c over the initial object I in $\mathcal{M}$NF, define $\mathtt{SeekSat}(c) = \mathtt{Fix}(\mathtt{Sat}(c))$. For a condition over P in $\mathcal{M}$NF, define $\mathtt{Sat}$ and $\overline{\mathtt{Sat}}$ as follows:

$$\begin{aligned}
\mathtt{Sat}(\exists a) &= \mathtt{if}\ \neg\exists a\ \mathtt{then}\ \textstyle\bigcup_{P\hookrightarrow C'\hookrightarrow C=a, C'\ncong C}\{\langle P \hookrightarrow C' \Rightarrow C\rangle\}\ \mathtt{fi}\\
\mathtt{Sat}(\exists(a,c)) &= \textstyle\bigcup_{P\hookrightarrow C'\hookrightarrow C=a}\{\langle P \hookrightarrow C' \Rightarrow C\rangle\}; \mathtt{Sat}(c)\\
\mathtt{Sat}(\wedge_{j\in J} c_j) &= \mathtt{while}\ (\neg\wedge_{j\in J} c_j)\ \mathtt{do}\ \ ;_{j\in J}\mathtt{if}\ \neg c_j\ \mathtt{then}\ \mathtt{Fix}(\mathtt{Sat}(c_j))\ \mathtt{fi}\ \ \mathtt{od}
\end{aligned}$$

where $;_{j\in\{1,\dots,n\}}\mathtt{P}_j = ((\mathtt{P}_1;\mathtt{P}_2);\dots;\mathtt{P}_n)$.

$$\begin{aligned}
\overline{\mathtt{Sat}}(\exists a) &= \mathtt{if}\ \exists a\ \mathtt{then}\ \mathtt{Abort}\ \mathtt{fi}\\
\overline{\mathtt{Sat}}(\exists(a,c)) &= \mathtt{while}\ \exists(a,c)\ \mathtt{do}\ \ \langle\langle P \overset{a}{\hookrightarrow} C \Rightarrow C\rangle, c\rangle; \overline{\mathtt{Sat}}(c)\ \ \mathtt{od}\\
\overline{\mathtt{Sat}}(\wedge_{j\in J} c_j) &= \textstyle\bigcup_{j\in J}\{\overline{\mathtt{Sat}}(c_j)\}
\end{aligned}$$

$\mathtt{Sat}(\mathrm{true}) = \mathtt{Skip}$, $\overline{\mathtt{Sat}}(\mathrm{true}) = \mathtt{Abort}$, $\mathtt{Sat}(\neg c) = \overline{\mathtt{Sat}}(c)$, and $\overline{\mathtt{Sat}}(\neg c) = \mathtt{Sat}(c)$.

In the case of $\mathtt{Sat}$, existential statements correspond to an expansion of existing substructures (if necessary): Given a morphism $P \hookrightarrow G$, the program $\mathtt{Sat}(\exists a)$ non-deterministically extends any partial occurrence C' to C, provided $\exists a$ is not already satisfied. Similarly, given a morphism $P \hookrightarrow G$, the program $\mathtt{Sat}(\exists(a,c))$ non-deterministically extends any partial occurrence C' to C and subsequently applies $\mathtt{Sat}(c)$ on that occurrence. Moreover, conjunction corresponds to an iterated random sequentialization until a solution is found (this iteration may not terminate). The completeness of $\mathtt{Fix}(\mathtt{Sat}(c))$ implies that the execution order of the subprograms c_j is irrelevant for the overall problem, and it suffices to consider just one sequentialization. Negation corresponds to a switch to the complementary $\overline{\mathtt{Sat}}$, and no computation is necessary in the case of true.

For the complementary $\overline{\mathtt{Sat}}$, the (non)satisfiability of a basic existential statement $\exists a$ is just checked: If $\exists a$ is satisfied, the computation is ended and a depth-first interpreter would backtrack. For a nested existential statement $\exists(a,c)$, an occurrence of C that does satisfy c is selected in the hope that a subsequent application of $\mathtt{Sat}(c)$ yields a result in which C does not satisfy c (this iteration may not terminate). Conjunction corresponds to nondeterministic choice between alternatives: only one subcondition has to be dissatisfied such that the negation of conjunction becomes satisfied. Negation corresponds to a switch to the complementary $\mathtt{Sat}$, and the computation is ended in the

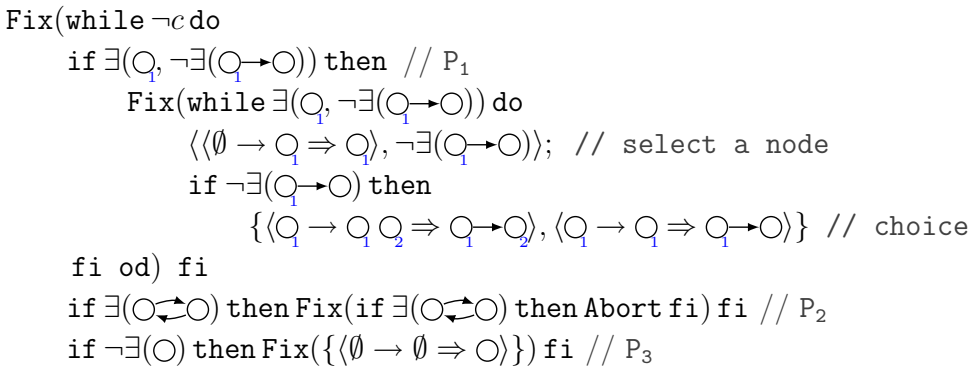
case of true.

Remark 3.3 For abbreviated conditions, the construction of $\mathtt{Sat}$ and $\overline{\mathtt{Sat}}$ is extended as follows:

$$\begin{array}{ll|ll}
\mathtt{Sat}(\forall(a,c)) & = \overline{\mathtt{Sat}}(\exists(a,\neg c)) & \overline{\mathtt{Sat}}(\forall(a,c)) & = \mathtt{Sat}(\exists(a,\neg c)) \\
\mathtt{Sat}(\mathrm{false}) & = \mathtt{Abort} & \overline{\mathtt{Sat}}(\mathrm{false}) & = \mathtt{Skip} \\
\mathtt{Sat}(c \Rightarrow d) & = \mathtt{Sat}(\neg c \vee d) = \{\overline{\mathtt{Sat}}(c), \mathtt{Sat}(d)\} & \overline{\mathtt{Sat}}(c \Rightarrow d) & = \overline{\mathtt{Sat}}(\neg c \vee d) \\
\mathtt{Sat}(\vee_{j\in J}\, c_j) & = \overline{\mathtt{Sat}}(\wedge_{j\in J}\, \neg c_j) & \overline{\mathtt{Sat}}(\vee_{j\in J}\, c_j) & = \mathtt{Sat}(\wedge_{j\in J}\, \neg c_j)
\end{array}$$

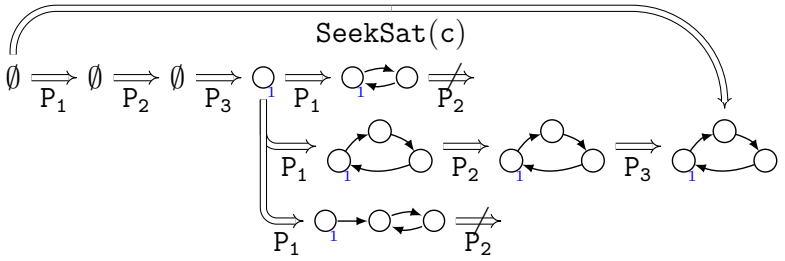
Remark 3.4 $\mathtt{Sat}(c)$ is a program of finite size for every condition c in $\mathcal{M}$NF: As we consider only finite conjunctions and disjunctions of conditions and the number of all decompositions $P \hookrightarrow C' \hookrightarrow C$ is finite (a consequence of a finite number of $\mathcal{M}$-matches), all program sets are finite and the sequentialization in case of $\mathtt{Sat}(\wedge_{j\in J}\, c_j)$ is of finite length.

Example 3.5 (satisfiable condition) Consider the following graph condition $c = \forall(\bigcirc_1, \exists(\bigcirc_1 \to \bigcirc)) \wedge \neg\exists(\bigcirc \rightleftarrows \bigcirc) \wedge \exists(\bigcirc)$ expressing "All nodes have an outgoing edge, there exists no cycle of length two and there is a node". The program $\mathtt{SeekSat}(c)$ is:



A fragment of the semantics of $\mathtt{SeekSat}(c)$ is depicted below by representing each morphism with its codomain (all depicted morphisms have I as do-

main/interface).



There exists some $G \in \mathbf{Graph}$ such that $\langle \mathrm{id}_I, i_G, \mathrm{id}_I \rangle \in [\![\mathtt{SeekSat}(c)]\!]$, hence c is satisfiable.

The rule sets in the cases of $\mathtt{Sat}(\exists a)$ and $\mathtt{Sat}(\exists(a,c))$ are minimal. For every rule left out, there is a satisfiable condition for which `SeekSat` is not complete anymore. We show this exemplarily for the cases $P \ncong C'$ and $C' \ncong C$ of $\mathtt{Sat}(\exists(a,c))$

Example 3.6 (minimality) Assume, it is possible to exclude the case $C' \ncong C$ for $\mathtt{Sat}(\exists(a,c))$, i.e. redefine $\mathtt{Sat}(\exists(a,c)) = \bigcup_{P \hookrightarrow C' \hookrightarrow C = a, C' \ncong C} \{\langle P \hookrightarrow C' \Rightarrow C \rangle\}; \mathtt{Sat}(c)$. Then the condition

$$c_1 = \exists(\bigcirc, \exists(\bigcirc^{0})) \wedge \exists(\bigcirc, \exists(\bigcirc^{1})) \wedge \neg\exists(\bigcirc^{0}\ \bigcirc^{1})$$

is satisfied by the graph $\overset{0}{\circlearrowright}\overset{1}{\circlearrowright}$, but there is no $\langle \mathrm{id}_I, m^*, \mathrm{id}_I \rangle \in [\![\mathtt{SeekSat}(c_1)]\!]$. Assume, it is possible to exclude the case $P \ncong C'$ for $\mathtt{Sat}(\exists(a,c))$, i.e. redefine $\mathtt{Sat}(\exists(a,c)) = \bigcup_{P \hookrightarrow C' \hookrightarrow C = a, P \ncong C'} \{\langle P \hookrightarrow C' \Rightarrow C \rangle\}; \mathtt{Sat}(c)$. Then the condition

$$c_2 = \exists(\bigcirc, \exists(\bigcirc^{0})) \wedge \exists(\bigcirc, \exists(\bigcirc^{1})) \wedge \neg\exists(\overset{0}{\circlearrowright}\overset{1}{\circlearrowright})$$

is satisfied by the graph $\bigcirc^{0}\ \bigcirc^{1}$, but there is no $\langle \mathrm{id}_I, m^*, \mathrm{id}_I \rangle \in [\![\mathtt{SeekSat}(c_2)]\!]$.

Fact 3.7 (monotonicity) `Sat` *and* $\overline{\mathtt{Sat}}$ *are monotone: for every condition* c *in* $\mathcal{M}$*NF, for every* $\langle m', m^*, \mathrm{id} \rangle \in \mathtt{Fix}(\mathtt{Sat}(c))$ $[\mathtt{Fix}(\overline{\mathtt{Sat}}(c))]$ *there is a total morphism* $x\colon \mathrm{codom}(m') \to \mathrm{codom}(m^*)$ *in* $\mathcal{M}$ *from the codomain of* m' *to the codomain of* m^* *such that* $x \circ m' = m^*$.

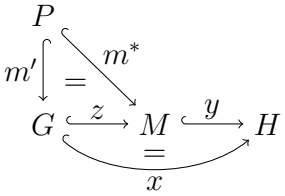
Proof As every condition c is in $\mathcal{M}$NF, every morphism $a\colon P \to C$ in every subcondition $\exists a$ and $\exists(a,c)$ is in $\mathcal{M}$. Consequently, `Sat` and $\overline{\mathtt{Sat}}$ are programs over non-deleting rules with total interface (see Fact 2.15). □

The proof of Theorem 3.2 is based on the following lemma.

Lemma 3.8 (`Sat` and $\overline{\mathtt{Sat}}$) *Let* $\mathrm{id}\colon P \to P$ *be the identity over* P. *For each condition* c *over an object* P, $\mathtt{Sat}(c)$ *and* $\overline{\mathtt{Sat}}(c)$ *are programs that, with respect to the satisfiability problem, are*

(correct) $\forall m', m^* \in \mathcal{M}.\ \langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{Fix}(\mathtt{Sat}(c))]\!]$ *implies* $m^* \models c$,
$\forall m', m^* \in \mathcal{M}.\ \langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{Fix}(\overline{\mathtt{Sat}}(c))]\!]$ *implies* $m^* \not\models c$,

(complete) $\forall m', x \in \mathcal{M}.\ x \circ m' \models c$ *implies* $\exists m^*, y, z \in \mathcal{M}.\ \langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{Fix}(\mathtt{Sat}(c))]\!]$, $m^* = z \circ m'$ *and* $x = y \circ z$.
$\forall m', x \in \mathcal{M}.\ x \circ m' \not\models c$ *implies* $\exists m^*, y, z \in \mathcal{M}.\ \langle m', m^*, \mathrm{id}\rangle \in [\![\mathtt{Fix}(\overline{\mathtt{Sat}}(c))]\!]$, $m^* = z \circ m'$ *and* $x = y \circ z$.



Completeness means, if a given $\mathcal{M}$-morphism m' can be extended by a $\mathcal{M}$-morphism x to satisfy c, there is a transition from m' to a $\mathcal{M}$-morphism m^* ($m^* = z \circ m'$ for some $\mathcal{M}$-morphism z) and m^* lies exactly along the way to $x \circ m'$ ($x = y \circ z$) preserving the possibility to eventually yield $x \circ m'$. While this does not imply a guarantee to reach $x \circ m'$ at all, we can reach a possibly smaller morphism, namely m^* that also satisfies the subcondition c (see correctness).

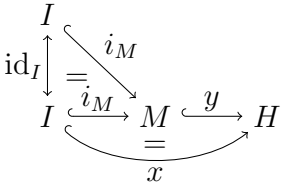
Proof of Lemma 3.8. See long version at [1]. □

Proof of Theorem 3.2. Without loss of generalization c is a condition over the initial object I in $\mathcal{M}$-normal form (see Fact 2.3). As $\mathtt{SeekSat}(c)$ is defined as $\mathtt{Fix}(\mathtt{Sat}(c))$, the correctness and completeness of $\mathtt{SeekSat}(c)$ are special cases of the correctness and completeness of $\mathtt{Fix}(\mathtt{Sat}(c))$. More precisely, as $m' = \mathrm{id}_I \colon I \leftrightarrow I$ and any morphism from the initial object is in $\mathcal{M}$ and c is a condition over I, the correctness of $\mathtt{Fix}(\mathtt{Sat}(c))$ reduces to

$$\langle \mathrm{id}_I, i_M, \mathrm{id}_I\rangle \in [\![\mathtt{Fix}(\mathtt{Sat}(c))]\!] \text{ implies } i_M \models c \text{ implies } M \models c,$$

and the completeness of $\mathtt{Fix}(\mathtt{Sat}(c))$ reduces to

$$H \models c \text{ implies } i_H \models c \text{ implies } \exists i_M.\ \langle \mathrm{id}_I, i_M, \mathrm{id}_I\rangle \in [\![\mathtt{Fix}(\mathtt{Sat}(c))]\!].$$



□

A consequence of the completeness of Theorem 3.2 is that termination implies a decision of the problem.

Corollary 3.9 *If* $\texttt{SeekSat}(c)$ *terminates and* $\nexists M.\ \langle \mathrm{id}_I, i_M, \mathrm{id}_I\rangle \in [\![\texttt{SeekSat}(c)]\!]$, *then* c *is not satisfiable.*

`SeekSat` is guaranteed to terminate for a certain fragment of conditions. Hence it is able to decide the satisfiability problem for this subclass.

Let Cond be the set of all conditions and let BCond be the $\exists a$*-fragment* of non-nested existential conditions, i.e. Boolean formulas over *basic* conditions $\exists a$.

Theorem 3.10 *For the* $\exists a$*-fragment of* Cond, `SeekSat` *is guaranteed to terminate.*

A consequence of Theorem 3.10 and Corollary 3.9 is the decidability of BCond.

Corollary 3.11 *For the* $\exists a$*-fragment of* Cond, `SeekSat` *decides the satisfiability problem.*

Let $\mathrm{BCond}^{\setminus\wedge}$ be the set of all non-nested existential conditions without conjunction. The proof of Theorem 3.10 is based on the following property: For all conditions $c \in \mathrm{BCond}^{\setminus\wedge}$ that do not contain conjunction, for all tuples $\langle m', m^*, \mathrm{id}\rangle$ in the semantics of $\texttt{Fix}(\texttt{Sat}(c))$, the satisfiability of all conditions in $\mathrm{BCond}^{\setminus\wedge}$ is preserved from m' to m^*, or the satisfiability of c is guaranteed from m^*.

Lemma 3.12 *For all conditions* $c \in \mathrm{BCond}^{\setminus\wedge}$, *for all* $\langle m', m^*, \mathrm{id}\rangle \in [\![\texttt{Fix}(\texttt{Sat}(c))]\!]$,

$$\begin{aligned} &\forall d \in \mathrm{BCond}^{\setminus\wedge}.\ m' \models d \ \ \textit{implies}\ \ m^* \models d, \\ \textit{or}\quad &\forall x \in \mathcal{M}\ \textit{with}\ x \circ m^* \in \mathcal{M}.\ x \circ m^* \models c. \end{aligned}$$

Proof of Lemma 3.12. See long version at [1]. □

Proof of Theorem 3.10. Let $c \in \mathrm{BCond}$ in conjunctive normal form, i.e. $c = \wedge_{j\in J} c_j$ and $c_j \in \mathrm{BCond}^{\setminus\wedge}$ for each $j \in J = \{1, \ldots, n\}$. The program $\texttt{SeekSat}(c)$ terminates if the while iteration of $\texttt{Sat}(\wedge_{j\in J}\ c_j)$ terminates. The iteration is guaranteed to terminate after at most n iterations, as in each iteration $\langle m_1, m_{n+1}, \mathrm{id}\rangle \in [\![\texttt{;}_{j\in J}\texttt{if}\ \neg c_j\ \texttt{then}\ \texttt{Fix}(\texttt{Sat}(c_j))\ \texttt{fi}]\!]$ with $\langle m_j, m_{j+1}, \mathrm{id}\rangle \in [\![\texttt{if}\ \neg c_j\ \texttt{then}\ \texttt{Fix}(\texttt{Sat}(c_j))\ \texttt{fi}]\!]$ for $j \in J$, there must be an index $k \in J$ in which the satisfiability of subcondition c_k is established and guaranteed from m^*. If such a step does not exists, the satisfiability of each condition c_ℓ, $\ell \in J$, is be preserved in each step $j \in J$ and for the whole

iteration. This case however can be excluded: by correctness of $\texttt{Fix}(\texttt{Sat}(c))$, this would mean $\wedge_{j\in J}\, c_j$ is already satisfied and contradict the test $\neg\wedge_{j\in J}\, c_j$ at the begin of the while iteration. □

In the following, we briefly state the implications of our results to the complementary tautology problem.

Definition 3.13 (tautology problem) For a given category $\mathcal{C}$, the *tautology problem* is the problem to decide for any given condition c, whether or not $\forall G \in \mathcal{C}.\ G \models c$.

Each instance of the tautology problem may be viewed as an instance of the satisfiability problem, by negating both the input condition, as well as the answer. However, in contrast to positives answers, negative answers of a satisfiability algorithm may only be lifted to the tautology problem in case of termination and completeness. Otherwise, the incompleteness of the algorithm would correspond to unsoundness in the case of the tautology problem. A consequence of Theorem 3.2, Theorem 3.10 is the following:

Corollary 3.14 *For the* $\exists a$*-fragment of* Cond, `SeekSat` *decides the tautology problem.*

4 Implementation and Optimization

In this section, we want to discuss practical aspects concerning an implementation of `SeekSat` and further optimizations.

Neither a pure depth-first, nor a pure breadth-first evaluation of `SeekSat` is guaranteed to find a result for a satisfiable condition: A depth-first execution may take a wrong choice towards an infinite subtree of the transition system without results, a breadth-first evaluation is at least not possible on the level of programs, as the unfolded transition system tree may have an infinite degree at some points. Either a breadth-first evaluation on the level of transformation rules or a small results-first evaluation seems to be the best way to organize the search.

The main goals of any non-deterministic algorithm is to reduce the number of available choices and to minimize backtracking. Therefore, the number of "equivalent" matchings that lead to isomorphic results must be reduced. As application conditions restrict the number of matches, rules with interface K should be replaced by rules with smaller interface K' and a positive existential application condition, if possible. In context of external interfaces, however, this seems to be only viable for the case $\texttt{Sat}(\exists a)$. Moreover, isomorphism checks should be applied to avoid unnecessary recomputations and seem ap-

propriate especially in context of a breadth-first or size-based evaluation.

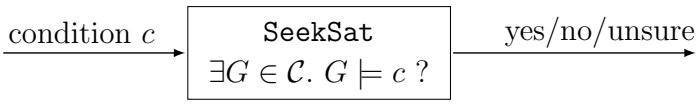
In case of conjunction, disjunction and for all program sets in general, the order in which subprograms are executed is free to choose. Known heuristics [15] should be applied here to determine an order in which viable choices are tried first and while others are suppressed until a later point. The propositional structure of conditions may be even used to rule out certain choices and to prune whole branches of the search tree. For the rule sets in case of $\exists a$ and $\exists(a, c)$, at least an initial idea is to order the rules by the number of elements they will introduce, with the aim to try to introduce as few elements as possible.

Before `SeekSat` is constructed for a condition c, the condition should be brought into normal form and optimized by a set of straightforward substitutions, e.g., $\exists(a, \exists b)$ may be substituted by $\exists(b \circ a)$.

In general, the information flow within programs should be improved, e.g., to avoid subsequent double checks of conditions. E.g., in case of $\texttt{Sat}(\wedge_{j\in J}\, c_j)$, the use of variables would bring some improvement:

```
while ∧_{j∈J} var_j do
    foreach j ∈ J do
        if ¬c_j then Fix(Sat(c_j)); foreach j ∈ J do var_j := false od fi;
        var_j := true
od  od
```

For practical purposes, artificial bounds may be introduced to yield a correct and terminating, but incomplete algorithm such that for every condition c, a positive answer will imply the satisfiability of c for some object $G \in \mathcal{C}$, but a negative answer will not always imply the absence of such an object. These restrictions could be based on CPU time, object's sizes, or the length or width of the search tree, and should preferably apply only for nested conditions $c \notin \mathrm{BCond}$.

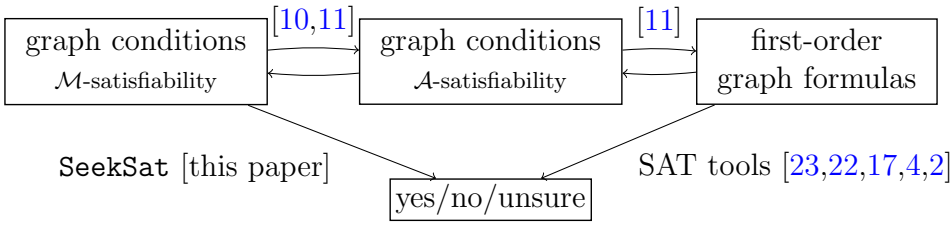


5 Related Concepts

In this section, we try to relate `SeekSat` to algorithms for the satisfiability problem (SAT) of first-order formulas. Before that, we briefly review the connection of first-order formulas and high-level conditions and discuss the

main differences between SAT algorithms on high-level conditions and first-order formulas.

In [11], transformations between graph conditions and first-order logic on graphs are considered, similar to [20]. For directed, labeled graphs, both concepts are expressively equivalent. The proof is based onto two steps: First, there are transformations between $\mathcal{M}$-satisfiable conditions (this definition) and $\mathcal{A}$-*satisfiable* conditions [10], i.e. conditions with a semantics in which the morphisms required to be present or absent (p, q in the definition) are arbitrary. Second, there are transformations between $\mathcal{A}$-satisfiable graph conditions and graph formulas, relating the semantics of formulas and conditions: on the one hand assignments of variables to a structure representing a graph, on the other hand arbitrary morphisms from the graphs in the condition to a tested graph. Note, the transformations of the first step are high-level, those of the second step are graph-specific.



In case of directed, labeled graphs, the translation into first-order formulas allows to use existing tools to solve the satisfiability problem of conditions, and may form the basis of an evaluation of `SeekSat` and its implementation. Still, the point of `SeekSat` is to make a translation of the problem unnecessary by providing an implementation of a SAT algorithm for any category satisfying the assumptions of Section 2.

The main differences between algorithms on high-level conditions and formulas are the following: high-level algorithms become structure-specific once they are "instantiated" for a given category. While SAT algorithms for general first-order logic necessarily consider arbitrary structures (and have to be restricted by a set of axioms to a target structure, which adds to the complexity of the problem), `SeekSat` will, by definition, only consider objects of the given category $\mathcal{C}$. Another difference is represented by $\mathcal{M}$- and $\mathcal{A}$-satisfiability: distinct elements in $\mathcal{M}$-satisfiable conditions are mapped onto distinct elements in the domain. In formulas, it remains open if the values of variables are equal or distinct, unless it is explicitly stated. Finally, an algorithm on conditions can and should use the fact that conditions make quantifications and statements in bulks. In this sense, conditions may have a lower logical complexity when compared to their translations in first-order logic.

Most first-order SAT solvers, including the most successful ones [18] such as Darwin [2] and Paradox [4] are based on *finite model building*, like their idols Mace2 [16], Mace4 [17], Falcon [22] and SEM [23]. Most of these tools (except Darwin) approach the satisfiability problem by translating it, for a given domain size, into a decidable SAT problem of either propositional logic or at least ground clauses with equality. This has the advantage of using existing implementations of well-known (propositional) SAT algorithms, such as the dominating Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6] and its derivatives, thus benefiting from years of experience and know-how. However, the translation phase is usually associated with a significant blow-up: Generating all ground instances over a domain of size n for a clause with v variables will yield n^v instances alone [17]. Also, the problem has to be solved again and again for increasing domain sizes, while only few tools are capable to reuse earlier results.

In contrast, `SeekSat` contains no such translation. Nevertheless, `SeekSat` seems, to some degree, related to the family of *enumeration* algorithms that are based on tree search and splitting, like the DPLL algorithm. `SeekSat` is based on a tree search where internal nodes correspond to partial solutions (morphisms), branches are choices (partitioning the search space), and leaf nodes are complete results or deadends. Instead of *splitting*, i.e. the process of branching by selecting a propositional variable x from a formula and assigning true and false, respectively, `SeekSat` will either skip, modify the morphism by adding elements to its codomain (positive statement) or backtrack (negative statement), depending on the satisfaction of the considered subcondition by the current morphism. While currently not the case, `SeekSat` can be made aware of the propositional structure of a condition to exclude whole branches of the search tree without losing results, as discussed in Section 4. This should strenghten the above relation.

Recently, the Model Evolution Calculus [3,2] was described, which lifts the propositional DPLL procedure to first-order logic. Similar to `SeekSat`, the split rule of the ME calculus is restricted to positive literals (the model evolves only in case of positive statements). Where `SeekSat` uses morphisms to apply a rule, the ME calculus uses unification. Like `SeekSat`, the ME calculus is shown to be correct and complete. It is claimed that the ME calculus can decide the Bernays-Schönfinkel ($\exists\forall$) fragment of first-order logic. The SAT solver Darwin is an implementation of the ME calculus [2] and was among the best solvers at the CADE 2007 [18].

6 Conclusion

We have presented a non-deterministic algorithm for the satisfiability problem of high-level conditions. It was shown that the algorithm is correct and complete, thus it is not guaranteed to terminate in general. A fragment of conditions was identified, namely $\exists a$-fragment of conditions, for which the termination of the algorithm was proved. Consequently, the algorithm can decide the satisfiability problem as well as the complementary tautology problem for this subclass. We have discussed certain aspects concerning an implementation and its optimization. For practical purposes, the algorithm can be converted into a non-complete, but terminating algorithm. The algorithm was formally described by using programs over transformation rules with external interfaces.

Further topics include

- an investigation, whether or not the presented algorithm is (directly) portable to conditions with arbitrary satisfiability [10,11] (conditions can no longer assumed to be in $\mathcal{M}$NF and one may require rules with $K \to R$ not in $\mathcal{M}$),
- a systematic study of rules with external interfaces,
- an algorithm for approximating the tautology problem (such an algorithm will yield results in some instances for which `SeekSat` does not terminate),
- an implementation of `SeekSat`,
- further comparison with existing first-order satisfiability algorithms and tools, such as Darwin [2], Mace4 [17] and SEM [23].

References

[1] `http://formale-sprachen.informatik.uni-oldenburg.de/pub/eindex.html`.

[2] Baumgartner, P., A. Fuchs and C. Tinelli, *Implementing the model evolution calculus*, International Journal on Artificial Intelligence Tools **15** (2006), pp. 21–52.

[3] Baumgartner, P. and C. Tinelli, *The model evolution calculus*, in: *Proc. 19th International Conference on Automated Deduction (CADE)*, LNAI (LNCS) **2741** (2003), pp. 350–364.

[4] Claessen, K. and N. Sörensson, *New techniques that improve MACE-style finite model finding*, in: *Proc. CADE-19 Workshop on Model Computation (MODEL)*, 2003.

[5] Courcelle, B., *Graph rewriting: An algebraic and logical approach*, in: *Handbook of Theoretical Computer Science*, volume **B**, Elsevier, Amsterdam, 1990 pp. 193–242.

[6] Davis, M., G. Logemann and D. Loveland, *A machine program for theorem-proving*, Commun. ACM **5** (1962), pp. 394–397.

[7] Ehrig, H., K. Ehrig, A. Habel and K.-H. Pennemann, *Theory of constraints and application conditions: From graphs to high-level structures*, Fundamenta Informaticae **74** (2006), pp. 135–166.

[8] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," EATCS Monographs of Theoretical Computer Science, Springer-Verlag, Berlin, 2006.

[9] Ehrig, H., A. Habel, J. Padberg and U. Prange, *Adhesive high-level replacement systems: A new categorical framework for graph transformation*, Fundamenta Informaticae **74** (2006), pp. 1–29.

[10] Habel, A. and K.-H. Pennemann, *Satisfiability of high-level conditions*, in: *Graph Transformations (ICGT'06)*, LNCS **4178** (2006), pp. 430–444.

[11] Habel, A. and K.-H. Pennemann, *Correctness of high-level transformation systems relative to nested conditions*, 2008, submitted.

[12] Habel, A., K.-H. Pennemann and A. Rensink, *Weakest preconditions for high-level programs*, in: *Graph Transformations (ICGT'06)*, LNCS **4178** (2006), pp. 445–460, a long version is available as technical report.

[13] Habel, A. and D. Plump, *Computational completeness of programming languages based on graph transformation*, in: *Proc. Foundations of Software Science and Computation Structures*, LNCS **2030** (2001), pp. 230–245.

[14] Heckel, R. and A. Wagner, *Ensuring consistency of conditional graph grammars*, in: *SEGRAGRA'95*, ENTCS **2**, 1995, pp. 95–104.

[15] Kumar, V., *Algorithms for constraint satisfaction problems: A survey*, AI Magazine **13** (1992), pp. 32–44.

[16] McCune, W., *MACE 2.0 reference manual and guide*, Tech. Memo ANL/MCS-TM-249, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (2001).

[17] McCune, W., *Mace4 reference manual and guide*, Tech. Memo ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (2003).

[18] Pfenning, F., editor, "Proc. 21th International Conference on Automated Deduction (CADE)," LNAI (LNCS) **4603**, Springer, 2007.

[19] Plump, D. and S. Steinert, *Towards graph programs for graph algorithms*, in: *Graph Transformations (ICGT'04)*, LNCS **3256** (2004), pp. 128–143.

[20] Rensink, A., *Representing first-order logic by graphs*, in: *Graph Transformations (ICGT'04)*, LNCS **3256** (2004), pp. 319–335.

[21] Trakhtenbrot, B. A., *The impossibility of an algorithm for the decision problem on finite classes (In Russian)*, Doklady Akademii Nauk SSSR **70** (1950), pp. 569–572, english translation in: Nine Papers on Logic and Quantum Electrodynamics, *AMS Transl. Ser. 2*, 23:1–5, 1963.

[22] Zhang, J., *Constructing finite algebras with FALCON*, Journal of Automated Reasoning **17** (1996), pp. 1–22.

[23] Zhang, J. and H. Zhang, *SEM: A system for enumerating models*, in: *Proc. International Joint Conferences on Artificial Intelligence (IJCAI)*, **1** (1995), pp. 298–303.