Giving a Model-Based Testing Language a Formal Semantics via Partial MAX-SAT

Bernhard K. Aichernig, Christian Burghard

Giving a Model-based Testing Language a Formal Semantics via Partial MAX-SAT

Bernhard K. Aichernig[1][0000−0002−3484−5584] and Christian Burghard[1,2] ✉

[1] *Institute of Software Technology*,
Graz University of Technology, Graz, Austria
{aichernig,burghard}@ist.tugraz.at
[2] AVL List GmbH, Graz, Austria
christian.burghard@avl.com ✉

Abstract. Domain-specific Languages (DSLs) are widely used in model-based testing to make the benefits of modeling available to test engineers while avoiding the problem of excessive learning effort. Complex DSLs benefit from a formal definition of their semantics for model processing as well as consistency checking. A formal semantics can be established by mapping the model domain to a well-known formalism. In this paper, we present an industrial use case which includes a mapping from domain-specific models to Moore Machines, based on a Partial MAX-SAT problem, encoding a predicative semantics for the model-to-model mapping. We show how Partial MAX-SAT solves the frame problem for a non-trivial DSL in which the non-effect on variables cannot be determined statically. We evaluated the performance of our model-transformation algorithm based on models from our industrial use case.

Keywords: Partial Moore Machines · Model Transformation · Consistency Checking · Formal Semantics · Frame Problem · Partial MAX-SAT.

1 Introduction

Model-based Testing (MBT) has emerged as a widely adopted practice for the verification of industrial systems [20,22,27]. Following this development, the AVL List GmbH is testing its portfolio of automotive measurement devices via an MBT approach based on a textual domain-specific modeling language called MDML. In our previous work [8,9], we designed the MDML language and modeling environment to provide our test engineers with an intuitively understandable model-based testing tool.

In a more recent development stage, we incorporated a model transformation from MDML models to Partial Moore Machines as the initial step of our model-based testing toolchain. This transformation to a well-defined formalism provides MDML with an operational semantics. In the current version of our toolchain, it is largely based on a *Partial MAX-SAT* (PMSAT) problem, encoding predicative semantics. We chose a SAT-based implementation for our model transformation to cope with the potentially complex interplay of specialized language features.

```
1  public statevar Mode {Standby, Measure} = Standby;
2  private statevar Timer {Off, On} = Off;
3  input Action {STBY, SMES, STIM};
4
5  given Mode = Standby {
6    when Action = SMES then Mode -> Measure;
7  }
8  given Mode = Measure {
9    when Action = STBY then Mode -> Standby;
10   given Timer = On when 15 sec elapsed then Mode -> last;
11 }
12 when Action = STIM then Timer -> On;
13 when Mode -> Standby then Timer -> Off;
14 fallback {
15   when Action = any then DoNothing;
16 }
```

Listing 1.1: Example MDML model

Before giving formal definitions, we briefly describe the syntax and semantics of MDML informally by the example model in Listing 1.1, which we will use for explanatory purposes throughout this work. Each MDML model starts with the definition of one or more *state variables* (Lines 1-2), including their name, value domain, initial value and a `public` or `private` classifier which indicates if the value of this variable is visible to an outside observer. This is followed by the definition of at least one `input` channel (Line 3), defining one or more input symbols through which the system can be controlled from the outside. The rest of the model consists of a hierarchy of rules, defining changes to state variables dependent on the current state variable values and the received input.

Beyond this basic structure, MDML incorporates several specialized language features which facilitate the modeling of specific behaviors or allow for a very concise model representation, e.g. `last`-*transitions* (Listing 1.1, Line 10), *secondary actions* (Line 13) or `fallback` *blocks* (Lines 14-16), all of which will be explained over the course of this work. These special features complicate the semantic evaluation of MDML models and, as our model-based testing toolchain evolved, the need for a more explicit model representation arose. A graph-based representation of the modeled state machine - e.g. in the form of a Moore Machine, would have a beneficial impact on further stages of our MBT toolchain, e.g test case generation [23], visualization [7] and tracing. A Partial Moore Machine generated from the MDML model in Listing 1.1 can be seen in Figure 1.

The semantics of MDML state machine models as collections of state variables makes them subject to the *frame problem* - i.e. the problem of reasoning about system variables which are implicitly assumed to remain unchanged during an operation. The use of PMSAT is dictated by specific MDML language constructs - the aforementioned *secondary actions* - which significantly complicate a purely SAT-based solution of the frame problem. Lines 9 and 10 in Listing 1.1, for example, each specify a transition from `Mode` = `Measure` to `Mode` = `Standby`. However, the secondary action in Line 13 states that, whenever the `Mode` changes to `Standby`, the `Timer` must change to `Off`. This is a condition on the post-state of a transition, depending on another property of the post-state. In Figure 1, the result can be seen as the two transitions labeled `STBY` which are not loops.

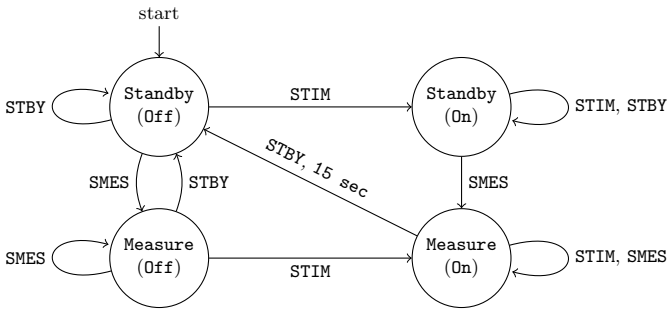

Fig. 1: Partial Moore Machine representation of the system represented in Listing 1.1. The values of `private` state variables, which are not part of the output, are displayed in brackets.

To the best of our knowledge, we are the first to provide a formal semantics to a domain-specific language via Partial MAX-SAT. Especially our treatment of the frame problem seems to be novel. We also contribute some experiences on the industrial use of a PMSAT-based model transformation. Defining an operational semantics for a front-end modeling language also has the effect of closely relating model transformation to consistency checking. We show how, through a small alteration, our model transformation setup can be used to uncover semantic inconsistencies within MDML models.

The rest of this paper is structured as follows: Section 2 introduces some necessary preliminaries, Section 3 outlines the syntax of the MDML language and Section 4 defines a formal semantics based on those syntax elements. Sections 5 and 6 explain how a model transformation and a consistency check can be implemented using this formal semantics. Section 7 presents the results of a runtime evaluation as well as our experience with the implementation. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2 Preliminaries

2.1 Measurement Device Modeling Language

The *Measurement Device Modeling Language* (MDML) was developed in close cooperation with AVL's test engineers to efficiently model the firmware state machines of measurement devices. The taxonomy of Utting et al. [27] classifies MDML as a *state-based* notation, which means that it specifies a system as a collection of variables, representing a snapshot of the internal state of the system, plus some operations that modify those variables through pre- and postconditions. The syntax of MDML allows for many degrees of freedom in the encoding of a specific state machine behavior and each engineer is able to develop a modeling style that suits his or her individual needs. After the test models have been established, we automatically generate test cases from them using various

mutation-based, structural and random coverage metrics [8]. These test cases are then converted to C# code and deployed to our test automation system. However, the aforementioned syntactic freedom of MDML is paid for by an increased difficulty in model processing. Although MDML is an effective input language, it would benefit from a more explicit model representation for all further uses in the model-based testing toolchain.

2.2 SAT, MAX-SAT and Partial MAX-SAT

A *Boolean satisfiability* (SAT) *problem* inquires, if a given Boolean formula can be fulfilled. While SAT problems are NP-complete, a series of algorithms and implementations ("SAT solvers") have been developed which can solve such problems in a (relatively) efficient and highly scalable manner [4]. When run on a Boolean formula $f(b_1, \ldots, b_n)$ on the variables b_1 to b_n, a SAT solver either returns a solution $S : [1, n] \to \mathbb{B}$, or the message `UNSAT` if f is not satisfiable. SAT solvers usually require formulas to be given in *conjunctive normal form* (CNF), i.e. as a conjunction of clauses, each consisting of a disjunctions of Boolean literals. However, any Boolean formula can be rewritten to CNF. The *MAX-SAT* problem is related to SAT. Here, the solver tries to maximize the number of satisfied clauses in a given CNF-formula. A further variant of the MAX-SAT problem is called *Partial MAX-SAT* (PMSAT) [10]. Here, the clauses are partitioned into a set of *hard* clauses (Φ^H_{SAT}), which must be satisfied in any case, and a set of *soft* clauses (Φ^S_{SAT}), of which a maximum number must be satisfied. Both MAX-SAT and PMSAT can be viewed as optimization problems with a cost function equal to the number of unfulfilled (soft) clauses.

2.3 Moore Machines

Definition 1 (Moore Machine). *A Moore Machine is a tuple* $\langle Q, q_0, I, O, \delta, \lambda \rangle$ *with a finite set of states* Q*, an initial state* $q_0 \in Q$*, a set of input symbols* I*, a set of output symbols* O*, a transition function* $\delta : Q \times I \to Q$ *and an output function* $\lambda : Q \to O$*.*

Definition 2 (Partial Moore Machine). *A Partial Moore Machine (PMM) is a generalization of a Moore Machine with a partial function* $\delta : Q \times I \nrightarrow Q$*.*

2.4 Frame Problem

The frame problem is a rather old problem that was first formulated by McCarthy and Hayes in a discussion of philosophical aspects of AI [17]. It is the problem of expressing the assumption in First-Order-Logic, that all system variables, which are not explicitly stated to be changed during an operation, remain unchanged. In classical specification methods, such as Z [25], VDM [13] and B [1], the set of changed variables for a given operation is known at the time of specification. Such a partial variable assignment can easily be transformed into a total assignment by assigning all remaining variables their current value. In this

way, the frame problem can be solved on a syntactic level. The Unified Theory of Programming (UTP) [2,11] makes this transformation from partial to total assignments explicit. Separation Logic [21] leverages this a-priori knowledge in order to reason about programs with a high number of state-defining variables.

3 MDML Syntax

The syntax of MDML is given by the following production rules:

$$
\begin{aligned}
Model &::= \mathcal{V}_S^+\ C^+\ Root_M\ Root_F^? \\
\mathcal{V}_S &::= (\texttt{public} \mid \texttt{private})\ \texttt{statevar}\ ID_v\ \{D_v\}\ \texttt{=}\ \iota_v\ \texttt{;} \\
D_v &::= x\ (\texttt{,}\ x)^* \\
x &::= ID \\
\iota_v &::= x \\
C &::= \texttt{input}\ ID_c\ \{I_c\}\texttt{;} \\
I_c &::= x\ (\texttt{,}\ x)^* \\
Root_M &::= Node^+ \\
Node &::= G \mid W \mid T \\
G &::= \texttt{given}\ Rule_G\ (\texttt{and}\ Rule_G)^*\ (Node \mid \{Node^+\}) \\
Rule_G &::= ID_v\ ((\ \texttt{=}\ \mid\ \texttt{!=}\ )\ x \mid \texttt{not}^?\ \texttt{in}\ \{x\ (\texttt{,}\ x)^*\}) \\
W &::= \texttt{when}\ Rule_W\ (\texttt{or}\ Rule_W)^*\ (Node \mid \{Node^+\}) \\
Rule_W &::= Action_I \mid \tau \mid Action_S \\
Action_I &::= ID_c\ ((\ \texttt{=}\ \mid\ \texttt{!=}\ )\ x \mid\ \texttt{=}\ \texttt{any} \mid \texttt{not}^?\ \texttt{in}\ \{x\ (\texttt{,}\ x)^*\}) \\
\tau &::= INT\ (\texttt{ms} \mid \texttt{sec} \mid \texttt{min})\ \texttt{elapsed} \\
Action_S &::= ID_v\ \texttt{->}\ x \\
T &::= \texttt{then}\ Rule_T\ (\texttt{and}\ Rule_T)^*\texttt{;} \\
Rule_T &::= ID_v\ \texttt{->}\ x \mid ID_v\ \texttt{->}\ \texttt{last} \mid \texttt{DoNothing} \\
Root_F &::= \texttt{fallback}\ \{Node^+\}
\end{aligned}
$$

In the above grammar, ID maps to an identifier terminal and INT to an integer terminal. In the rest of this paper, we will re-use some of the non-terminal symbols to denote the set of all their instantiations within a given syntactically correct MDML model - i.e. the set of all *state variables* $\mathcal{V}_S$, the set of all *input channels* C, the set of all `given` statements G, the set of all `when` statements W, the set of all `then` statements T and the set $Node$ for the union of the latter three. In contrast, we use D_v as the set of domain values and ι_v as the initial value for a given state variable v, as well as I_c as the set of input values for a given input channel c. Let I_τ be the set of all instantiations of τ and let $RNode = Node \cup \{Root_M, Root_F\}$. With $\mathcal{P}$ signifying the powerset, we define the function $children : RNode \to \mathcal{P}(Node)$ which maps a non-terminal instantiation

to the set of all instantiations of *Node* on the right side of its production rule. We also define $parent : Node \to RNode$ as the inverse of *children* and $ancestors : Node \to \mathcal{P}(RNode)$ as the transitive closure of *parent*. We illustrate all three functions on Lines 14-16 of Listing 1.1:

$$children(\texttt{when Action = any then DoNothing;}) = \{(\texttt{then DoNothing;})\}$$
$$parent(\texttt{when Action = any then DoNothing;}) = Root_F$$
$$ancestors(\texttt{then DoNothing}) = \{Root_F, (\texttt{when Action = any} \ldots)\}$$

Using these functions, we formulate the additional syntactic rule that for each `then` statement $t \in T$, $ancestors(t)$ must contain exactly one `when` statement from W. There are also some type restrictions to be considered. For a given state variable $v \in \mathcal{V}_S$, ι_v must be an element of D_v and for each occurrence of ID_v on the right side of a production rule together with an x, x must be an element of D_v. We also define all different D_v to be disjoint sets of state variable values, even if shared value IDs occur. For each input channel $c \in C$, each occurrence of ID_c together with an x on the right side of a production rule, x must be a member of I_c. All I_c are pairwise disjoint the same way as all D_v.

4 MDML Semantics

With a clearly defined syntax in place, we now define a formal semantics for MDML. By mapping MDML models to Partial Moore Machines (PMMs), we first establish an operational semantics. This mapping, in turn, relies upon a Boolean predicative semantics.

4.1 Operational Semantics

Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be the set of all model variables, including but not limited to $\mathcal{V}_S$. Further, let $D_\times = D_{v_1} \times \cdots \times D_{v_n}$ and $D_\cup = D_{v_1} \cup \cdots \cup D_{v_n}$. Let $public : \mathcal{V} \to \mathbb{B}$ be *true*, iff the argument is an element of $\mathcal{V}_S$ and declared `public`. Let $\mathcal{V}_P = \{v_{P1}, \ldots, v_{Pm}\} = \{v \in \mathcal{V}_S \mid public(v)\}$ and let $D_{P\times}$ be defined analogous to $D_\times$, on $\mathcal{V}_P$, instead of $\mathcal{V}_S$. We define the semantics of MDML models by mapping them to PMMs in the following way:

$$Q \subseteq D_\times \tag{1}$$
$$q_0 =_{df} [\iota_{v_1}, \ldots, \iota_{v_n}] \tag{2}$$
$$I =_{df} I_\tau \cup \bigcup_{c \in C} I_c \tag{3}$$
$$O \subseteq D_{P\times} \tag{4}$$
$$\lambda(q) =_{df} [eval(q, v_{P1}), \ldots, eval(q, v_{Pm})] \tag{5}$$

Other than externally triggered transitions, MDML also supports time-triggered transitions (see Listing 1.1, Line 10). Rather than giving MDML a full-fledged

Timed-Automata semantics, we simply treat time triggers (I_τ) in the same way as external inputs, in the sense of "the external controller lets time pass" (see Equation 3). This abstraction is not without its shortcomings but it is sufficient for our MBT purposes. As shown by Equation 1, the state space of the PMM is a subset of the space of possible valuations for the variables in $\mathcal{V}$. We define $eval : D_\times \times \mathcal{V} \to D_\cup$ to retrieve the value of a given variable in a given state. Finally, we define the transition function δ, based on the predicate $transition : D_\times \times I \times D_\times \to \mathbb{B}$:

$$\forall q \in Q, i \in I, q' \in D_\times : \delta(q, i) = q' \wedge q' \in Q \Leftrightarrow transition(q, i, q') \tag{6}$$

The remainder of this section will be dedicated to defining $transition(q, i, q')$ based on the contents of a given MDML model. We first break it down into its individual semantic components and finally relate them to the syntactic elements of MDML.

4.2 Predicative Semantics

The *transition* predicate is dependent on the *primary action* predicate *pa*, the *secondary action* predicate *sa*, the *history* predicate *hist* and the *framing* predicate *fra*:

$$transition\,(q, i, q') \Leftrightarrow pa(q, i, q') \wedge sa(q, i, q') \wedge hist\,(q, q') \wedge fra(q, i, q') \tag{7}$$

For all Boolean predicates introduced in this and the next section, we assume an implicit dependence on q, i, and q'. In these definitions, we use $\underline{v}$ and $\underline{v}'$ as a shorthand for the assignment of $v \in \mathcal{V}$ in q and q', respectively.

Primary Action. A primary action is the immediate response of the state machine to an external input or a time-trigger (see $Action_I$ and τ in the grammar definition, as well as Lines 6, 9, 10, 12 and 15 in Listing 1.1), encoded in the rule hierarchy of the MDML model. This hierarchy is composed of `given` statements, encoding conditions on q, `when` statements, encoding conditions on i and `then` statements, encoding conditions on q'. A `then` statement is *enabled* if the conditions of all its superordinate `given` statements evaluate to *true* and it is *triggered* if the condition of its superordinate `when` statement evaluates to *true*. A predicative definition of "enabled" and "triggered" will follow in Section 4.3. Let $T_P \subseteq T$ be the set of `then` statements which are both enabled and triggered. In a well-formed MDML model, the set of primary actions T_P has either zero or one elements. In the case of zero elements, the MDML model does not specify a transition for the specific pair of pre-state and input. In this case, we define $transition(q, i, q') =_{df} false$ and $\delta(q, i)$ remains undefined. If T_P contains more than one element, the MDML model is malformed and an inconsistency error is reported. For all further considerations, we assume $T_P = \{t_P\}$.

$$pa(q, i, q') =_{df} upd(t_P) \tag{8}$$

The *update* predicate $upd : T \to \mathbb{B}$ encodes the effect of a `then` statement on q' and will also be defined in Section 4.3.

Secondary Action. A secondary action is a value change of a variable v which is conditioned on a value change of a different variable v_w, as exemplified by Listing 1.1, Line 13. The effect is akin to that of a *change-trigger* in UML [19]. Secondary actions are encoded as `when` statements with a trigger rule of the form $Action_S ::= ID_{v_w}$ `->` x_w. Let $T_S \subseteq T \times Action_S$ be the set of pairs of all enabled `then` statements and rules of their superordinate `when` statements of the form $Action_S$. We also restrict T_S to those pairs which share the same root node as t_P. The secondary action predicate is given as:

$$sa(q,i,q') =_{df} \bigwedge_{\langle (ID_{v_w} \texttt{->} x_w), t\rangle \in T_S} \Big(\big(\underline{v_w} \neq x_w \wedge \underline{v'_w} = x_w \big) \rightarrow upd(t) \Big) \tag{9}$$

As can be seen from the above definition, the updates of each `then` statement t are performed if v_w simultaneously changes from a value other than x_w to x_w.

History Variable Maintenance. MDML offers a language construct called `last`-transition (see Listing 1.1, Line 10), where a state variable is reset to its most recent previous value. This cannot be accomplished without adding additional model variables on a semantic level. Let $\mathcal{V}_H \subseteq \mathcal{V}_S$ be the set of all state variables v for which a `then` rule $Rule_T ::= ID_v$ `->` `last` exists. We define a new set of state variables called *history* variables $\mathcal{H}$, which are bijectively associated with the variables in $\mathcal{V}_H$. For each $v \in \mathcal{V}_H$, we have an associated $h_v \in \mathcal{H}$ with $D_v = D_{h_v}$ and $\iota_v = \iota_{h_v}$. We can now complete the definition of the set of model variables as $\mathcal{V} = \mathcal{V}_S \cup \mathcal{H}$. The variables in $\mathcal{H}$ will be used to store the previous value of their counterparts in $\mathcal{V}_H$. This is accomplished by the history maintenance predicate:

$$hist\,(q,q') =_{df} \bigwedge_{v \in \mathcal{V}_H} \Big(\big(\underline{v'} = \underline{v} \rightarrow \underline{h'_v} = \underline{h_v} \big) \wedge \big(\underline{v'} \neq \underline{v} \rightarrow \underline{h'_v} = \underline{v} \big) \Big) \tag{10}$$

Framing. The predicates defined above clearly define $\underline{v'}$ for some $v \in \mathcal{V}_S$ and leave it undefined for others. If the $\underline{v'}$ is not defined by either the primary or the secondary action, it shall remain unchanged:

$$fra(q,i,q') =_{df} \bigwedge_{v \in \mathcal{V}_S} \neg def(v) \rightarrow \underline{v'} = \underline{v} \tag{11}$$

$$def(v) =_{df} \exists x \in D_v : pa(q,i,q') \wedge sa(q,i,q') \vdash \underline{v'} = x \tag{12}$$

The helper predicate $def : \mathcal{V}_S \rightarrow \mathbb{B}$ evaluates to *true* iff a unique value for $\underline{v'}$ can be formally deduced (expressed by the symbol $\vdash$) from the primary and/or secondary action predicates. The effect of the framing predicate can be observed in Listing 1.1, Lines 5-7, where transitions from states with $\underline{\texttt{Mode}} = \texttt{Standby}$ to $\underline{\texttt{Mode}}' = \texttt{Measure}$ are specified. These transitions exist regardless of the value of $\underline{\texttt{Timer}}$ and trigger no secondary actions. Therefore, the framing predicate will cause $\underline{\texttt{Timer}}' = \underline{\texttt{Timer}}$ for each resulting transition. This results in the two transitions in Figure 1 which are labeled `SMES` and are not self-transitions.

4.3 Rule Hierarchy

As previously stated, we express the semantics of `given` and `when` statements via the predicates *en* ("enables") and *tr* ("triggers"), which are defined on the syntactic elements of the rule hierarchy:

$$
\begin{aligned}
en(Root_M) &\Leftrightarrow_{df} true \\
en(G) &\Leftrightarrow_{df} en(parent(G)) \wedge en(Rule_{G1}) \\
&\qquad \wedge \cdots \wedge en(Rule_{Gk}) \\
en(ID_v \texttt{ = } x) &\Leftrightarrow_{df} \underline{v} = x \\
en(ID_v \texttt{ != } x) &\Leftrightarrow_{df} \underline{v} \neq x \\
en(ID_v \texttt{ in } \{x_1, \ldots, x_k\}) &\Leftrightarrow_{df} \underline{v} \in \{x_1 \ldots, x_k\} \\
en(ID_v \texttt{ not in } \{x_1, \ldots, x_k\}) &\Leftrightarrow_{df} \underline{v} \notin \{x_1 \ldots, x_k\} \\
en(W) &\Leftrightarrow_{df} en(parent(W)) \\
en(T) &\Leftrightarrow_{df} en(parent(T)) \\
en(Root_F) &\Leftrightarrow_{df} \nexists\, t \in T_P : Root_M \in ancestors(t)
\end{aligned}
$$

The main root of the rule hierarchy ($Root_M$) is enabled by default. Each `given` statement inherits the enabledness of its parent and imposes its own conditions ($Rule_G$) on q in the form of equality, inequality and set exclusion and inclusion of state variable values $\underline{v}$ in q. Both `when` and `then` statements inherit enabledness from their parent statements. The `when` statements in Lines 12 and 13 of Listing 1.1, for example, are direct children of $Root_M$ and always enabled while those in Lines 6, 9 and 10 are conditioned upon the value of the state variable `Mode` in q. We complete the definition of *en* by examining the semantics of the `fallback` root $Root_F$, which is enabled when the main rule hierarchy specifies no primary action. The `fallback` mechanism was designed to provide the users of MDML with an easy way to incorporate "exception handling" in their models or make their models input-complete. Secondary actions ($Action_S$) do not rely upon this predicate and evaluate to *false*. The semantics of `when` statements is encoded in the predicate *tr*:

$$
\begin{aligned}
tr(Root_M) &\Leftrightarrow_{df} false \\
tr(Root_F) &\Leftrightarrow_{df} false \\
tr(G) &\Leftrightarrow_{df} tr(parent(G)) \\
tr(W) &\Leftrightarrow_{df} tr(Rule_{W1}) \vee \cdots \vee tr(Rule_{Wk}) \\
tr(ID_c \texttt{ = } x) &\Leftrightarrow_{df} i = x \\
tr(ID_c \texttt{ != } x) &\Leftrightarrow_{df} i \in I_c \setminus \{x\} \\
tr(ID_c \texttt{ in } \{x_1, \ldots, x_n\}) &\Leftrightarrow_{df} i \in \{x_1, \ldots, x_k\} \\
tr(ID_c \texttt{ not in } \{x_1, \ldots, x_n\}) &\Leftrightarrow_{df} i \in I_c \setminus \{x_1, \ldots, x_k\} \\
tr(ID_c \texttt{ = any}) &\Leftrightarrow_{df} i \in I_c
\end{aligned}
$$

$$
\begin{aligned}
tr(\tau) &\Leftrightarrow_{df} i = \tau \\
tr(Action_S) &\Leftrightarrow_{df} false \\
tr(T) &\Leftrightarrow_{df} tr(parent(T))
\end{aligned}
$$

Unlike the value of en, tr starts out as *false* on both roots. If the current input i satisfies a trigger rule $Rule_W$ of a `when` statement, the statement gets triggered. Here, $Rule_W$ can again be based on equality or inequality, as well as set inclusion and exclusion relative to the input alphabet I_c of a given input channel c. However, since i is not guaranteed to be an element of I_c, the conditions are formulated in a type-safe manner. Alternatively, primary actions can be triggered by time triggers τ. Analogous to en, `given` and `then` statements inherit the tr value of their parent node. The effect of a `then` statement on q' is encoded by the update predicate upd:

$$
\begin{aligned}
upd(T) &\Leftrightarrow_{df} upd(Rule_{T1}) \wedge \cdots \wedge upd(Rule_{Tk}) \\
upd(ID_v \texttt{ -> } x) &\Leftrightarrow_{df} \underline{v}' = x \\
upd(ID_v \texttt{ -> last}) &\Leftrightarrow_{df} \underline{v}' = \underline{h_v} \\
upd(\texttt{DoNothing}) &\Leftrightarrow_{df} \bigwedge_{v \in \mathcal{V}_S} \underline{v}' = \underline{v}
\end{aligned}
$$

The most common case of $Rule_T$ is the assignment of $\underline{v}'$ to a specific value x. Alternatively, the rule `DoNothing` encodes a self-transition ($\delta(q, i) = q$) and ID_v `->` `last` causes $\underline{v}'$ to assume the value that was held prior to $\underline{v}$.

5 Model Transformation

To transform a given MDML model into a PMM, we first obtain I and q_0 directly from the model. We then perform a closure on Q under δ for all $i \in I$. To do that, we state the *transformation problem* as obtaining a post-state q' for a given pre-state q and input i according to Equation 6. The transformation problem is a *Constraint Satisfaction Problem* (CSP), which we encode as a PMSAT problem and solve with an appropriate solver. We do, however perform a prefiltering step by computing T_P and T_S via imperative algorithms and therefore exclude all irrelevant model elements from the CSP. This was done to further both performance and ease of implementation.

5.1 PMSAT-Encoding of the Transformation Problem

The assignments of the MDML model variables in both pre- and post-state become the variables of the CSP:

$$
\mathcal{V}_{CSP} = \bigcup_{v \in \mathcal{V}} \{\underline{v}, \underline{v}'\} \tag{13}
$$

There are multiple ways to encode a CSP into SAT. For the transformation problem, we follow the *direct encoding* method described by Walsh [28]. The

encoding of CSP variables with multi-value domains into Boolean SAT variables is accomplished by:

$$\mathcal{V}_{SAT} = \left\{ b_{\underline{v}x} \in \mathbb{B} \mid \underline{v} \in \mathcal{V}_{CSP} \wedge x \in D_v \wedge \left(b_{\underline{v}x} = true \Leftrightarrow \underline{v} = x \right) \right\} \tag{14}$$

Although there are more concise encodings, the direct encoding was again chosen to ease implementation. The MDML model in Listing 1.1, for example, would produce eight Boolean variables, for `Mode=Standby`, `Mode=Measure`, `Timer=On` and `Timer=Off` in both q and q'. This encoding requires additional clauses to ensure that each CSP variable is assigned exactly one value. The valuation in the pre-state q is known a-priori and can be directly evaluated as $x = eval(q, v)$ and all Boolean variables encoding q can be set to a fixed value:

$$pre(q) = \bigwedge_{v \in \mathcal{V}} \left(\underline{v} = x \wedge \bigwedge_{\tilde{x} \in D_v \setminus \{x\}} \underline{v} \neq \tilde{x} \right) \tag{15}$$

The valuation in the post-state q is as of yet unknown and only restricted by the known variable domains.

$$post(q') = \bigwedge_{v \in \mathcal{V}} \left(\left(\bigvee_{x \in D_v} \underline{v}' = x \right) \wedge \bigwedge_{x_1 \in D_v, x_2 \in D_v \setminus \{x_1\}} (\underline{v}' = x_1 \rightarrow \underline{v}' \neq x_2) \right) \tag{16}$$

Leaving the conversion to CNF implicit, the full set of hard clauses for the PMSAT problem emerges as:

$$\Phi^H_{SAT} = pa(q, i, q') \wedge sa(q, i, q') \wedge hist\,(q, q') \wedge pre(q) \wedge post(q') \tag{17}$$

Up to this point, $fra(q, i, q')$ has not been represented in terms of a first-order logic formula. In CSP problems, framing axioms explicitly ensure that every variable change has a defined cause[14]. However, this would require us to establish an explicit cause-and-effect relationship between variable changes to ensure that each secondary action is directly or indirectly caused by the primary action. Otherwise, the solution might contain groups of secondary actions which are causing each other - i.e. forming causal loops. Instead of using framing axioms, we decided to formulate the transformation problem in PMSAT and model $fra(q, i, q')$ by means of soft clauses:

$$\Phi^S_{SAT} = \bigwedge_{v \in \mathcal{V}_S} \underline{v}' = \underline{v} \tag{18}$$

Theorem 1. *If the SAT problem Φ^H_{SAT} has a solution,*[3] *the solution to the PM-SAT problem $\langle \Phi^H_{SAT}, \Phi^S_{SAT} \rangle$ satisfies transition (q, i, q').*

[3] Note that the reverse implication is trivially fulfilled, since PMSAT can only pick the optimum from all possible solutions of the underlying SAT problem.

Proof. The structure of the proof is as follows: First, we find a solution which minimizes the cost function of the PMSAT problem. Then, we confirm that this solution fulfills the framing predicate and is a valid solution to the SAT problem. Let $\mathcal{V}_d \subseteq \mathcal{V}_S$ be the set of all state variables which are uniquely defined, i.e. which are assigned the same value in all solutions of the pure SAT problem. Let $\mathcal{V}_f = \mathcal{V}_S \setminus \mathcal{V}_d$ be the set of all state variables which are not uniquely defined. Following Equation 18, the cost function of the PMSAT problem can be reformulated in the following way, intuitively showing the global minimum:

$$E(q') = c + \sum_{v \in \mathcal{V}_f} \begin{cases} 0 & \textbf{iff } \underline{v}' = \underline{v} \\ 1 & \textbf{otherwise} \end{cases} \tag{19}$$

Here, c is a constant value caused by the variables in $\mathcal{V}_d$. The post-state $q'_{\min}$ with $\forall v \in \mathcal{V}_f : \underline{v}' = \underline{v}$ yields the minimum cost value $E(q'_{\min}) = c$. We now examine the fulfillment of each predicate in Equations 7 and 17:

Primary action: The primary action predicate references q' solely via the predicate *upd*, which either does not reference any given $\underline{v}'$ or restricts it to a single value. Therefore, all variables referenced by the primary action must be element of $\mathcal{V}_d$ and $q'_{\min}$ cannot falsify this predicate.

Secondary action: The secondary action predicate references q' via the update predicate, as well as in the left-hand side of an implication. If a variable from $\mathcal{V}_d$ triggers a secondary action, all variables referenced by *upd* will be uniquely determined (see proof for primary action) and, in turn, be part of $\mathcal{V}_d$. The valuation of $\mathcal{V}_f$ in $q'_{\min}$, cannot trigger any secondary actions since $\underline{v}' = \underline{v}$ for all $v \in \mathcal{V}_f$. In other words, all variables in $\mathcal{V}_f$ trivially fulfill the predicate for $q'_{\min}$.

History: We assume that the assignment of all $h \in \mathcal{H}$ in $q'_{\min}$ satisfies the history predicate. All variables in $\mathcal{V}_S$ are either not referenced or referenced in the left-hand side of an implication and therefore cannot falsify the predicate.

Pre-State: The pre-state predicate does not reference q'.

Post-State: The post-state predicate establishes the domain for all $\underline{v}'$ but does not restrict them any further.

Framing: Let $\mathcal{V}_{dp} = \{v \in \mathcal{V}_S \mid \mathit{def}(v)\}$ and $\mathcal{V}_{fp} = \mathcal{V}_S \setminus \mathcal{V}_{dp}$. From the definition of *def* and $\mathcal{V}_d$, it follows that $\mathcal{V}_{dp} \subseteq \mathcal{V}_d$ and, consequently, $\mathcal{V}_f \subseteq \mathcal{V}_{fp}$. From the above proofs, it is evident that, if a given $\underline{v}'$ is not uniquely determined by the primary or secondary action, it also cannot be further determined by any other predicates. Therefore we get $\mathcal{V}_{fp} \subseteq \mathcal{V}_f$ and, consequently, $\mathcal{V}_{fp} = \mathcal{V}_f$. In $q'_{\min}$, we have $\underline{v}' = \underline{v}$ for all variables in $\mathcal{V}_f$, fulfilling the framing predicate.

With exception of the framing predicate, all other predicates are (either generally or specifically for $q'_{\min}$) independent of all variables in $\mathcal{V}_f$. Therefore $q'_{\min}$ must be a valid solution to the SAT problem.

□

6 Consistency Check

In order to be transformed into a Partial Moore Machine (PMM), an MDML model needs to be *well-formed*. To define the notion of well-formedness, we ex-

ploit an important duality between model transformation and consistency checking: An MDML model is well-formed, iff the transformation into a PMM succeeds. Or in other words, each failure mode of the transformation corresponds to a possible type of model error. To avoid redundancies in our implementation, we directly check the consistency of MDML models by attempting a model transformation. Errors in the MDML model may take one of the following forms:

- Syntax errors are handled directly by the textual editor.
- Domain errors, e.g. `when` c `=` x with $x \notin I_c$ are also handled by the editor.
- Explicit transformation errors, e.g. $|T_P| > 1$ are handled as software exceptions, thrown within the transformation algorithm.
- Contradictions within an MDML model can be discovered if $|T_P| = 1$ but $\nexists\ q' \in D_\times : transition\,(q, i, q')$. They will be the focus of this Section. Contradictions can be as obvious as `then Mode -> Standby and Mode-> Measure`, but also more subtle due to the involvement of secondary actions.
- Remaining gaps in the semantics definition - e.g. multiple time-triggered `then` statements enabled in the same pre-state - must be explicitly disallowed and prevented via dedicated consistency checking methods.

6.1 Inconsistency Detection via PMSAT

Assuming the transformation problem has no solution, despite having a unique primary action, we define the *checking problem* as detecting a set of model elements which causes the contradiction. The checking problem could be solved by computing an unsatisfiability core and removing any axiomatic clauses, e.g. from $pre(q)$, $post(q')$ or $hist\,(q, q')$. To avoid a further increase of requirements to the solver's capabilities, we instead formulate the checking problem again as a PMSAT problem. If a contradiction in the model prevents a solution of the transformation problem (Eq. 17), it is due to the presence of inconsistent *upd* predicates which only occur as part of *pa* and *sa*. To identify these contradictory predicates, we set up the checking problem $\langle \tilde{\Phi}^H_{SAT}, \tilde{\Phi}^S_{SAT} \rangle$ as a modified version of the transformation problem with *pa* and *sa* moved to the set of soft clauses:

$$\tilde{\Phi}^H_{SAT} = hist\,(q, q') \wedge pre(q) \wedge post(q') \tag{20}$$

$$\tilde{\Phi}^S_{SAT} = pa(q, i, q') \wedge sa(q, i, q') \tag{21}$$

In this way, the solver will remove the minimum number of clauses from *pa* and *sa* to produce a solution for q'. Each one of these unfulfilled clauses contains an update predicate (or part thereof) which contributes to the contradiction. By iterating through all solutions with the optimal cost value and backtracking the unfulfilled clauses of each solution to their syntactic element of origin, we obtain a set of update rules ($Rule_T$) which make the model inconsistent. These model elements are then marked as erroneous. Hence, we use PMSAT for debugging.

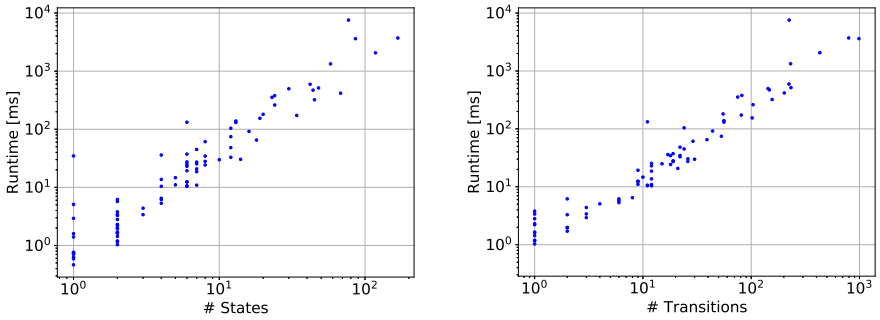

Fig. 2: Runtime evaluation of the model transformation with respect to the number of states $|Q|$ (left) and the number of transitions $|\delta|$ (right)

6.2 Warnings against Dead Code and Redundancy

For the consistency checking of MDML models, we define an *error* as an aspect of the model which makes it unsuitable for further processing - e.g. test case generation. All other issues which do not fall under this category are classified as *warnings*. As such, we regard aspects of the MDML model which we believe may not reflect the intentions of the user, e.g. unreachable or redundant MDML statements. To identify such model elements, we relate the PMM, which was obtained through the model transformation, back to the original MDML model. The set of unreachable statements - i.e. dead code - can be obtained as $Node_U = \{n \in Node | \nexists\, q \in Q : en(q, n)\}$ while the set of redundant `given` statements is obtained as $G_R = \{g \in G \mid \nexists\, q \in Q : en(q, parent(g)) \wedge \neg en(q, g)\}$.

7 Evaluation

We have implemented the previously described model transformation and consistency checking mechanism as part of our MDML IDE, using the SAT4J solver [4]. We have analyzed the runtime of the model transformation on 87 MDML models, taking the median of 10 repetitions for each model. Figure 2 displays the runtime measurements with respect to the number of states ($|Q|$), as well as the number of transitions ($|\delta|$) of the individual MDML models. The runtime is polynomial with respect to both the number of states and transitions, as indicated by the linear-shaped distributions in both doubly-logarithmic plots.

8 Related Work

Our solution to the frame problem is similar to that of Answer Set Programming (ASP) [15]. The frame rule in ASP states that, if a property holds in the pre-state and it cannot be derived that the property does not hold in the post-state, then it is assumed that the property holds in the post-state.

Mapping a DSL to a Moore Machine-based semantics has been done before [3,5,24]. However, to the best of our knowledge, we are the first to define a predicative semantics of a DSL based on PMSAT. Biere et al. [6] have mapped the Bounded Model Checking problem to SAT. However, their technique is concerned with verifying temporal properties while we aim to check model consistency, unreachability, and redundancy. Hwong et al. [12] have designed and implemented a mapping from the textual *State Machine Language* to a process algebraic language called mCRL2 in order to verify the highly complex control software of one of the experiments at CERN. They then use Bounded Model Checking to detect live-locks and ensure *strong connectedness*. Löding and Peleska [16] present an operational semantics for their *Timed Moore Automata* formalism, which they use for livelock detection and SAT-based test case generation. Mechtaev et al. have devised a tool called DirectFix [18] for program repairs based on PMSAT or, more specifically, Partial MAX-SMT. Similarly to our approach, Tien et al. [26] use PMSAT to find contradictory constraints in SysML models.

9 Conclusion

We have defined a formal semantics for the pre-existing domain-specific modeling language MDML. We split the definition into an operational semantics, relating MDML models to Partial Moore Machines, and a predicative semantics, defining the relation between pre- and post-states of each transition. We have designed and implemented a model transformation from MDML models to PMMs, based on a Partial MAX-SAT problem. To the best of our knowledge, we are the first to show how PMSAT can be used to solve the frame problem when an approach based on pure SAT would significantly complicate the solution. However, we kept parts of the implementation imperative wherever the cost of a SAT-based implementation outweighed the benefits. We prioritized ease of implementation over performance, since the test models in our domain are rather small. In theory, we would be able to eliminate the need for PMSAT by excluding Secondary Actions from the language design. However, they considerably simplify the implementation of certain behaviors, resulting in shorter models. We have further established a duality between model transformation and consistency checking for MDML models, enabling us to turn the transformation into a consistency check with little effort. Our future work may encompass the automatic repair of contradictory MDML models, perhaps in a similar manner as DirectFix [18].

Acknowledgements. Research herein was partially funded by the Austrian Research Promotion Agency (FFG), program "ICT of the Future", project number 867535 Hybrid Domain-specific Language User eXperience (HybriDLUX).

References

1. Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press (1996). https://doi.org/10.1017/CBO9780511624162

2. Aichernig, B.K., He, J.: Refinement and test case generation in UTP. Electron. Notes Theor. Comput. Sci. **187**, 125–143 (2007). https://doi.org/10.1016/j.entcs.2006.08.048
3. Alenljung, T., Lennartson, B., Hosseini, M.N.: Sensor graphs for discrete event modeling applied to formal verification of PLCs. IEEE Trans. Contr. Sys. Techn. **20**(6), 1506–1521 (2012). https://doi.org/10.1109/TCST.2011.2168607
4. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. JSAT **7**(2-3), 59–6 (2010), https://satassociation.org/jsat/index.php/jsat/article/view/82
5. Beyak, L., Carette, J.: SAGA: A DSL for story management. In: Danvy, O., Shan, C. (eds.) Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011. EPTCS, vol. 66, pp. 48–67 (2011). https://doi.org/10.4204/EPTCS.66.3
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
7. Burghard, C., Berardinelli, L.: Visualizing multi-dimensional state spaces using selective abstraction. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE (2020), accepted for publication.
8. Burghard, C.: Model-based Testing of Measurement Devices Using a Domain-specific Modelling Language. Master's Thesis at Graz University of Technology (2018)
9. Burghard, C., Stieglbauer, G., Korošec, R.: Introducing MDML - a domain-specific modelling language for automotive measurement devices. CEUR Workshop Proceedings **1711**, 28–31 (2016)
10. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings. pp. 252–265 (2006). https://doi.org/10.1007/11814948_25
11. Hoare, C.A.R., Jifeng, H.: Unifying theories of programming, vol. 14. Prentice Hall Englewood Cliffs (1998)
12. Hwong, Y., Keiren, J.J.A., Kusters, V.J.J., Leemans, S.J.J., Willemse, T.A.C.: Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. Sci. Comput. Program. **78**(12), 2435–2452 (2013). https://doi.org/10.1016/j.scico.2012.11.009
13. Jones, C.B.: Systematic software development using VDM. Prentice Hall International Series in Computer Science, Prentice Hall (1986)
14. Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M.: Global state checker: Towards SAT-based reachability analysis of communicating state machines. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013, co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, October 1st, 2013. CEUR Workshop Proceedings, vol. 1069, pp. 31–40. CEUR-WS.org (2013), http://ceur-ws.org/Vol-1069/06-paper.pdf
15. Lifschitz, V.: What is answer set programming? In: Fox, D., Gomes, C.P. (eds.) Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008. pp. 1594–1597. AAAI Press (2008), http://www.aaai.org/Library/AAAI/2008/aaai08-270.php
16. Löding, H., Peleska, J.: Timed moore automata: Test data generation and model checking. In: Third International Conference on Software Testing, Verification and

Validation, ICST 2010, Paris, France, April 7-9, 2010. pp. 449–458. IEEE Computer Society (2010). https://doi.org/10.1109/ICST.2010.60
17. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence. pp. 463–502. Edinburgh University Press (1969)
18. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. pp. 448–458. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.63
19. Object Management Group: OMG Unified Modelling Language. version 2.5.1. Tech. rep. (December 2017), https://www.omg.org/spec/UML/2.5.1/PDF
20. Peleska, J.: Industrial-strength model-based testing - state of the art and current challenges. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013. EPTCS, vol. 111, pp. 3–28 (2013). https://doi.org/10.4204/EPTCS.111.1
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
22. Sarma, M., Murthy, P.V.R., Jell, S., Ulrich, A.: Model-based testing in industry: a case study with two MBT tools. In: Zhu, H., Chan, W.K., Budnik, C.J., Kapfhammer, G.M. (eds.) The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa. pp. 87–90. ACM (2010). https://doi.org/10.1145/1808266.1808279
23. Schlick, R., Herzner, W., Jöbstl, E.: Fault-based generation of test cases from UML-models – approach and some experiences. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) Computer Safety, Reliability, and Security. pp. 270–283. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
24. Schuts, M., Hooman, J., Kurtev, I., Swagerman, D.: Reverse engineering of legacy software interfaces to a model-based approach. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznań, Poland, September 9-12, 2018. Annals of Computer Science and Information Systems, vol. 15, pp. 867–876 (2018). https://doi.org/10.15439/2018F64
25. Spivey, J.M.: Understanding Z : A specification language and its formal semantics. Ph.D. thesis, University of Oxford, UK (1985), http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.371571
26. Tien, T.N., Nakajima, S., Thang, H.Q.: Modeling and debugging numerical constraints of cyber-physical systems design. In: Thang, H.Q., Thanh, B.N., Do, T.V., Bui, M., Hong, S.N. (eds.) 4th International Symposium on Information and Communication Technology, SoICT '13, Danang, Viet Nam - December 05 - 06, 2013. pp. 251–260. ACM (2013). https://doi.org/10.1145/2542050.2542068
27. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability **22**(5), 297–312 (2012). https://doi.org/10.1002/stvr.456
28. Walsh, T.: SAT v CSP. In: International Conference on Principles and Practice of Constraint Programming. pp. 441–456. Springer (2000)