Abstract—Automatic verification of programs and computer systems with input variables represents a significant and well-motivated challenge. The case of Simulink diagrams is especially difficult, because there the inputs are read iteratively and the number of input variables is in theory unbounded. We apply the techniques of explicit model checking to account for the temporal (control) aspects of verification and use set-based representation of data, thus handling both sources of nondeterminism present in the verification. Two different representations of sets are evaluated in scalability with respect to the range of input variables. Explicit (enumerating) sets are very fast for small ranges but fail to scale. Symbolic sets, represented as first-order formulae are considered in scalability with respect to the range of input variables. Thus the proposed method allows complete automatic verification without the need to limit the nondeterminism of input.

I. INTRODUCTION

The primary motivation of this paper is the verification of Simulink diagrams against specifications given as linear temporal logic (LTL) formulae. The methodology we propose is more general and can accommodate a wider class of systems and models, but the research is guided by this concrete application. Consider the diagram below.

A Simulink diagram is an oriented graph, which can be interpreted as abstract syntax tree, except it may contain self-referencing cycles. The expression associated with a node is described by the tree underneath it, e.g. in our diagram $d'_1 = \min\{c_3, (¬sv_1 ∨ c_1 > sd_{13})?c_4 : d_1\}*$ $\int v(¬sv_1 ∨ c_1 > sd_{13})$.

Other than the arithmetic and logical operators there are 4 types of nodes: delays (box), imports (circle), outports (ellipse), and constants (plain). Only delays are self-referential and the semantics is that of a single element memory storage: whenever referred to, the delay node provides the stored value. That stored value $d$ is initiated to 0 and is replaced by $d'$ with every tick of the Simulink global clock.

Since the import nodes serve as inputs – providing a new, nondeterministic value after every tick – a Simulink diagram generates a transition system of the possible values stored in delays. Such a transition system represents the possible evolution of the diagram, describing its behaviour as it changes in time. Similarly to parallel and reactive programs, a developer may need to verify that the behaviour of a system complies with the specification. Consider for example a specification requiring that: whenever the import $sv_1$ is negative in three consecutive ticks, the value of $pm_2$ will eventually be positive. Such a specification cannot be expressed in terms of safety, i.e. as a reachability of bad states, because it requires a violating run would be infinite. Consequently, verification procedures that do not allow temporal specification, such as most implementations of symbolic execution [17], static analysis [13] or deductive verification [24], cannot be used in this case. Model checking [12], [25] allows verification against temporal properties, yet Simulink diagrams and similar systems with input variables pose another obstacle in the form of data-flow nondeterminism.

In order to answer such verification queries automatically, an algorithm must allow evaluation of Peano arithmetic expressions that form subtrees in the diagram. At the same time the input variables may be evaluated to an arbitrary number (which in practice is bounded, e.g. by the number of bits used to express the value) and each of these evaluations must be considered for the verification to be exhaustive. Standard symbolic approaches to model checking are either restricted to only a subset of Peano arithmetic [10] or are concerned with Boolean input variables [21]. Standard explicit approaches to model checking already suffer from the state space explosion caused by the control-flow nondeterminism and thus cannot scale to any considerable degree with the allowed ranges of input variables [3].

Hence the motivation of LTL verification of Simulink diagrams is sound given its potential to replace testing in the design phase of software (or systems) development. In this sense, the work of [3] succeeded only in part. In a tool-chain, and preceded by a tool to formalise pattern-based, human language requirements into LTL formulae, a model checker could provide similar results as testing for considerably smaller expenses. Yet the severely limited ranges of input variables the standard explicit-state model checking allows still enforce a test-like approach to correctness validation. Effectively, the developer still has to mechanically test (a smaller number of) instances of the same validation query with different (intervals of) input values.

In [2] the authors have proposed a combination of explicit and symbolic approaches to model checking, allowing verification of programs with input variables. Standard explicit model checker is accompanied with a symbolic representation of the set of possible evaluations which greatly reduces the state space. However, the implementation was rather experimental: sets of evaluations were stored explicitly and the employed modelling language DVE allowed reading from the input only once, at

Temporal Verification of Simulink Diagrams

Jiří Barnat, Petr Bauch, Vojtěch Havel
Masaryk University
the beginning. This work builds upon those results and extends them to allow temporal verification of Simulink diagrams.

**Contribution:** We have first formalised the process of model checking of Simulink diagrams, building on the concept of set-based reduction of the underlying state space, proposed in [2]. This reduction can significantly decrease the number of states, but does not lead to any reduction in the worst case. Also the representation used for the sets of variable evaluations determines the resulting complexity, which can differ exponentially for particular representations. We have implemented the set-based reduction using explicit sets as one representation and using bit-vector formulae as another representation. Explicit sets are exponential in both the number of variables represented and in their ranges, but allow simple implementation of *state matching*, i.e. deciding if two states represent the same evaluations. The representation using formulae is exponentially more succinct, but the state matching is a complex operation that requires deciding satisfiability of quantified bit-vector formulae.

Finally, the results of this paper show that (1) in the case of Simulink diagrams the set-based reduction is efficient, leading to exponentially smaller states spaces; (2) the representation using explicit sets allows fast verification but only for small ranges of variables; and (3) the representation using bit-vector formulae scales to arbitrary (bounded) ranges, though the state matching is resource expensive. In the terms of practical results, the proposed SAT-based verification is fully automated. Provided that the LTL specification and the Simulink model are available, the developer can demonstrate the correctness without any additional manual effort.

### A. Related Work

Firstly, there are three practical papers, that we are aware of, describing LTL verification of Simulink diagrams. Miller et al. [23] translate Simulink models into the synchronous data-flow language Lustre [16] and then into the native language of the symbolic model checker NuSMV [11]. NuSMV was then used to verify correctness of the model with respect to a set of specifications (mostly in CTL, but NuSMV can verify full LTL as well) using BDD-based (Binary Decision Diagrams see [21], for a classical approach) representation of the state space. Although it was not a limitation in the input diagrams they used, BDDs are not readily able to support complex data types with Peano arithmetic, at least not while preserving their efficiency, as we will discuss later in this subsection.

Similarly, Meenakshi et al. [22] translate Simulink diagrams directly to be verified by NuSMV. This work is more relevant because the diagrams used contained integer data types and full Peano arithmetic (NuSMV flattens integer types into Boolean types, but represents the stored values precisely). Unfortunately, we could not compare that approach with the one proposed in this paper, since the implementation is not available. Its efficiency can be only assumed from the reported results: when input variables were not bounded explicitly, i.e. allowed values between 0 and $2^{32}$, the verification time was almost a week; the bounds for reasonably fast verification had to be relatively small: between 0 and 60.

Explicit model checking was also used for verification of Simulink diagrams [8], but there the data-flow nondeterminism in the form of input variables had even greater detrimental effect. This is the only comparable implementation presently available and we provide detailed comparison between this purely explicit approach and the hybrid, control explicit—data symbolic approaches described in this paper in Section [IV].

From a theoretical point of view, the method proposed in this paper is effectively equivalent to symbolic execution extended with LTL verification. This combination was attempted before [8], but classical symbolic execution does not allow state matching and thus only a small subset (safety properties) of LTL was supported. For the same reason, i.e. the lack of support for LTL [19], we cannot compare our results with the Simulink Design Verifier, which is a part of the Matlab library allowing verification of assertion-based temporal (and other) properties.

A considerable amount of work pertained to allowing symbolic model checking to use more complex data types than Booleans. The problem is that BDDs grow exponentially in the presence of integer multiplication [9]. Similarly to our approach, symbolic model checking can also use first-order formulae directly, leading to SAT-based [6] and SMT-based [1] approaches, provided that the procedure is bounded in depth, and consequently incomplete. Various approaches to unbinding have been proposed, most recently the IC3 method [7], but these are rather theoretical, without implementation, or limited to weaker arithmetics, see e.g. the model checker Kind [15] based on $k$-induction.

## II. Preliminaries

Let us start by introducing the *theory of bit-vectors (BV)*, first-order theory where a bit width $q$ is associated with every term. The universe of this theory is natural numbers whose binary representation uses at most $q$ bits; we will denote this domain as $\text{bv}$/$q$. The signature of $\text{BV}$ subsumes Peano arithmetic – all operations are computed modulo $q$, as is common in many programming languages – and it also contains bit-level operations. The ability to precisely reason about computer arithmetic and the existence of efficient decision procedures [18] are the two main reasons why $\text{BV}$ is popular in automating both hardware and software verification. In the following we will assume notation similar to that of [18], i.e. a first-order evaluation $\nu$ maps a predicate $p$ to true: $\nu \models p$; a function $f$ is evaluated to $\nu(f)$; finally, $\nu \circ \{ x \mapsto y \}$ is $\nu$ except the variable $x$ is now mapped to $y$.

Let $X$ be a set of $\text{bv}$ variables. Then we can define LTL recursively as $\Psi := p_X \mid \neg \Psi \mid \Psi \land \Psi \mid X \cdot \Psi \mid \Psi \cup \Psi$, where $p_X$ is a $\text{BV}$ predicate over the variables from $X$. A parallel program with $t$ threads and with inputs, $\text{bv}$ variables $I$, can be modelled as a *transition system* $\mathcal{M} = (S, \rightarrow, s_0)$. There the states $S$ are triplets $((l_1, \ldots, l_t), \nu : X \rightarrow \text{bv}, p_X, I)$, $l_i$ are program locations of individual threads, $\nu$ is a $\text{BV}$ valuation of variables and $p_X \cup I$ a *path condition*, i.e. a predicate
stating the condition that would lead the execution to this state. The transition relation \( \rightarrow \subseteq S \times Act \times Con \times S \), where \( Act = \{ (x, f_{X \cup \mathcal{I}}) \mid x \in X \} \) and \( Con \) the set of predicates over \( X \), will be denoted as \( s \xrightarrow{(x, f_{X \cup \mathcal{I}})} s' \).

Let \( \pi_i \) be the standard projection functions, then a run of \( \mathcal{M} \) is an infinite sequence \( w = s_0, s_1, \ldots \), where for all \( i \in \mathbb{N} \): \( s_i \xrightarrow{(x, f)} s_{i+1} \) and for \( \hat{p} = p[\alpha/\pi(s_i)(\alpha)] \) it holds that \( \pi_1(s_{i+1}) = \pi_2(s_i) \wedge \hat{p} \) is satisfiable; finally we need to change the valuation of \( x \) such that \( \pi_2(s_{i+1}) < (x \mapsto \pi_2(s_i))(\pi_2(s_i)(\mathcal{I})) \). To see when a run \( w \) satisfies an LTL formula \( \Psi \), \( w \models \Psi \) and deeper description of LTL, consults for example [25].

Every LTL formula \( \Psi \) can be translated into a Büchi automaton \( A_\Psi = (S, \rightarrow, F) \), where \( \rightarrow \subseteq S \times Con \times S \) and \( F \subseteq S \) is the set of accepting states. For \( A_\Psi \) it holds that a run \( w \) of \( M \) satisfies \( \Psi \) if there exists a run \( w' = s'_1, s'_2, \ldots \) in \( A_\Psi \) such that for all \( i \in \mathbb{N} \), \( s'_i \rightarrow s'_{i+1} \) we have \( \pi_2(s_i) \vdash p \) and infinitely many states of \( w' \) are in \( F \). One can construct a product of \( M \) and \( A_\Psi \), where the only runs are those satisfying \( \Psi \). Then the states are still triplets only the sequence of program locations is longer by one, \( l_{i+1} \) denoting the state of \( A_\Psi \). It follows that deciding whether \( M \) satisfies \( \Psi \) can be reduced to locating cycles with accepting states in such a product. But under this definition, which is very similar to that of Lin [20], both sources of nondeterminism in a parallel program – the interleaving of threads and the evaluation of input variables – are represented explicitly. In symbolic execution [17], the transition system is unfolded (acyclic and infinite), \( S \) is thus a multi-set and the second component of a state is a set of possible evaluations, represented symbolically (every \( x \in X \) is associated with a \( BV \) function over \( \mathcal{I} \)). In [2], the authors have proposed a set-based reduction of \( M \), where two states are equal if they have the same sequence of program locations and their sets of evaluations are the same. Hence the set-based reduction extends the approach of symbolic execution by allowing the detection of accepting cycles, and consequently, verification of LTL properties.

**Example 1.** The figure below depicts a program with input

```
1: read a;
2: if (a > 5)
3:   a++;
4: else
5:   a--;
```

on the left and its associated transition system on the right. The set of evaluations (of the one variable \( a \)) is represented symbolically, as a function over input symbols from \( X \), here only \( l_1 \).

III. METHODS

We start by formalising the Simulink diagrams, presented in the introduction. There are several specific details of Simulink diagrams with respect to the general programs with inputs presented above, which may simplify the verification and thus need to be pointed out in detail. Also, until specified otherwise, the logical system used in our description is the quantifier-free fragment of \( BV \) with the bit-width \( q \), i.e. over sort \( bvq \) (although constants are written in the decimal system).

A state of a Simulink system is described by the evaluation of delay variables, \( X = \{ d_3, \ldots, d_m \} \), where their values for the next iteration are given by a function \( f_{X \cup \mathcal{I}} \), denoted as \( l_i \). There is a finite number of input variable templates, \( \mathcal{I} = \{ i_1, \ldots, i_n \} \), which can be read iteratively, but for each there are bit-vector constraints limiting its possible evaluation (represented by a predicate \( p_\mathcal{I} \), denoted as \( l_i \)). Finally, the complete description of a system also requires the output functions \( O = \{ \sigma_1, \ldots, \sigma_k \} \) (each of which is \( f_{X \cup \mathcal{I}} \)). On the side of the specification, we will abstract the given Büchi automaton to only the predicates \( p_{Y \cup \mathcal{I} \cup \mathcal{O}} \), labelling its transitions, \( \Phi = \{ \varphi_1, \ldots, \varphi_l \} \), and assume that the transition system, the order in which the predicates occur, is given implicitly. Hence the example below contains all the information needed for a Simulink diagram verification query \( Q = (X, \mathcal{I}, O, \Phi) \).

**Example 2.** Let \( Q_1 = (X, \mathcal{I}, O, \Phi) \), where

\[
\begin{align*}
X &= \{ d_1, d_2 \} \\
\mathcal{I} &= \{ i_1, i_2, i_3 \}
\end{align*}
\]

\[
\begin{align*}
\delta_1 : (i_1 + 1) \ast (i_2 + d_2) \\
\delta_2 : d_1 \\
\delta_3 : d_1 + i_3 \\
\end{align*}
\]

\[
\begin{align*}
\sigma_1 : \text{max}(i_1, i_2, i_3) \\
\sigma_2 : i \text{te } (i_1 > 7) d_1 d_2 \\
\sigma_3 : d_1 + i_3 \\
\end{align*}
\]

\[
\begin{align*}
\varphi_1 : \sigma_2 > 3 \lor i_1 \leq \sigma_1/2 \\
\varphi_2 : \sigma_1 = \sigma_2 \Rightarrow i_2 \leq 4
\end{align*}
\]

As mentioned above, the states of a Simulink transition system are sets of evaluations of the variables from \( X \). The initial state consists of a single evaluation \( \{ d_1 \rightarrow 0 \} \). Let \( \mathcal{I} = \{ x := \{ i_j \rightarrow y \} \mid x \models l_j \} \) be the \( |\mathcal{I}| \)-dimensional polyhedron of allowed input evaluations, for \( Q_1 \), it would be \( \{1..20\} \times \{3..5\} \times \{5..10\} \). Then, computing every transition consists of three steps:

1) compute the Cartesian product (of this state) with \( \mathcal{I} \) results in a set of evaluations of variables from \( X \cup \mathcal{I} \);
2) create a new state, initialised with the above set of evaluations, for every available specification formula \( \varphi \) (there can be more than one, hence the transition system may branch at this point) and remove those evaluations that do not satisfy \( \varphi \);
3) apply \( \delta \) functions to compute the new evaluations of \( X \).

Assuming that the computation is in state \( s \), represented as the evaluation of delay variables, and that the relevant transition is associated with specification \( \varphi \), then
\[ s' = \{ x := [d_i \mapsto v[\delta_i](v[\delta], v[\iota])] \mid v \in s \times \mathbb{I} \wedge x \models \varphi \} \]

is the resulting state. The system can reach a deadlock if there is no such \( v \in s \times \mathbb{I} \) for which \( [d_i \mapsto v[\delta_i](v[\delta], v[\iota])] \models \varphi \). The model checking process computes the graph of the Simulink computation, where identical states are unified, i.e. if a newly generated state \( s'' \) (as a successor of \( s \)) is found equal to an already existing state \( s' \), then \( s'' \) is discarded and \( s' \) is acknowledged as a successor of \( s \).

For the purpose of providing the user with counterexample traces, the states also contain an \([\mathbb{I}]\)-long sequence of inport values that satisfies the specification condition. But that is mostly easily implementable, and will only be mentioned if a significant change in the computation was necessary to incorporate the counterexample traces.

**Example 3.** We continue with the query \( Q_1 \) from Example 2 and simplify the notation by establishing two transformations. First, \( \bar{\varphi}_1 \) maps a set of evaluations of \( X \times \mathbb{I} \) to that subset where every evaluation satisfies \( \bar{\varphi}_i \), formally \( A \subseteq \text{bv}^{X \times \mathbb{I}} \mapsto \{ x \in A \mid x \models \bar{\varphi}_i \} \). Second, \( \Delta \) maps a set of the same evaluations to evaluations of \( X \), after the \( \delta \) functions have been applied, hence \( A \subseteq \text{bv}^{X \times \mathbb{I}} \mapsto \{ (d_i \mapsto x[\delta_i]) \mid x \in A \} \). Hence a part of the transition system of \( Q_1 \) is as follows:

\[
\begin{align*}
s_0 &= \Delta \circ \bar{\varphi}_1(s_0 \times \mathbb{I}) \\
n_0 &= (d_i \mapsto 0) \\
s_{01} &= \Delta \circ \bar{\varphi}_1(\Delta \circ \bar{\varphi}_1(s_0 \times \mathbb{I}) \times \mathbb{I}) \\
s_{02} &= \Delta \circ \bar{\varphi}_2(s_0 \times \mathbb{I}) \\
n_{012} &= \Delta \circ \bar{\varphi}_2(s_{01} \times \mathbb{I})
\end{align*}
\]

Concretely, the set \( s_0 \times \mathbb{I} \) has 360 elements. \( \bar{\varphi}_1(s_0 \times \mathbb{I}) \) and \( \bar{\varphi}_2(s_0 \times \mathbb{I}) \) have 63 and 315 elements, respectively; \( s_{01} \) has 12 elements and \( s_{02} \) has 45 elements. For example \( s_{01} = \{ (3,0), (4,0), (5,0), (6,0), (8,0), (9,0), (10,0), (12,0), (15,0), (16,0), (20,0), (25,0) \} \).

The process described so far produces a transition system, where the states are evaluations of delay variables and a location of the Büchi automaton, and the branching is given by that same automaton. Without going into details with respect to the product of program and its specification – where we went in the previous section – we will answer a verification query \( Q \) positively if there are no accepting cycles in the produced transition system, and negatively otherwise. The efficiency of answering a verification query, the complexity of Simulink model checking, is most noticeably influenced by the choice of how to represent the set of evaluations. That is the biggest contributor to the state space explosion, since, unlike in parallel programs, the control-flow nondeterminism is negligible in Simulink models.

**A. Verification with Explicit Sets**

The first representation considered in this paper stores the allowed evaluations in an explicit set, enumerating all possible combination of individual variable evaluations. This approach is closely related to that of [3] but improves on it in several aspects. Effectively, rather than repeating execution for every evaluation of both import and delay variables, we only represent the delay variables in an explicit evaluation; the import variables are stored purely symbolically, only as their constructing predicates \( \iota_j \), outside individual states. The above description of Simulink verification semantics justifies this optimisation, since the imports are only used for the evaluation of \( \Phi \) predicates from the specification. They are discarded during the \( \Delta \) transformation, and can be recovered using an inverse transformation, e.g. during counterexample generation.

The implementation itself is then a relatively straightforward extension of the purely explicit approach. The states enumerate every allowed evaluation and thus a separate access to each of these evaluations is permitted. Computing the Cartesian product with \( \mathbb{I} \) amounts to enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations. For the \( \bar{\varphi} \) and \( \Delta \) transformations, one iteratively considers enumerating all combinations of delay and import variable evaluations.

The need to iteratively read from inputs via the import variables complicates the otherwise relatively straightforward application of symbolic execution. Later we will show that the execution-based approach, where current values of variables are represented as symbolic functions over input variables, entails another complication in deciding state equality. In standard symbolic execution, the variables are initialised to an arbitrary value only once, at the beginning. Thus reading from imports requires to strengthen the standard symbolic execution to allow computation with potentially infinite number of input variables.

Henceforth, only the import variables are considered as first-order variables, the delays and outports are only functions in the \( \text{BV} \) theory (over import variables). The imports are history-dependent and thus one may label the symbols from \( \mathcal{I} \) to stand for the import variables, i.e. \( \mathcal{I}^0, \mathcal{I}^1, \mathcal{I}^2 \) and so on. Given that the only branching in the transition system is caused by the particular choice of a specification transition, the states can be represented as \( l \)-long sequences of numbers \( \rho = (r_1, \ldots, r_l) \), where \( l \) is the length of this computation. Each number in this sequence represents which specification formula
was selected in the respective branching of the computation. Indeed, the functions representing delays $\delta^l$, outports $\sigma^l$ and the specification predicates $\phi^l$ are unambiguously defined for each $l$ recursively as follows:

$$
\begin{align*}
\delta^0, \delta^{l+1} &= \delta[i_j/i_j^{l+1}, d_j/d_j^l], \\
\sigma^0, \sigma^{l+1} &= \sigma[i_j/i_j^{l+1}, d_j/d_j^l], \\
\phi^0, \phi^{l+1} &= \phi[i_j/i_j^{l+1}, d_j/d_j^l],
\end{align*}
$$

where $f[x/y]$ is the formula $f$ with every occurrence of $x$ replaced by $y$. Hence every formula at any point of the verification computation uses only the input variables decorated with history indices.

As mentioned above, the system reaches a deadlock if no value satisfies the specification formula. Let us assume that the computation is presently in a state represented by $\rho$ and the outgoing transition is labelled with formula $\phi_p$. Then checking whether this transition leads to a deadlock is equivalent to checking satisfiability of the following recursive path condition formula, where $\epsilon$ is an empty sequence:

$$
\begin{align*}
pc(\epsilon) &= \bigwedge_{1 \leq j \leq |I|} i_j^0, \\
pce(\rho, \rho') &= pc(\rho) \land \bigwedge_{1 \leq j \leq |I|} i_j^{l+1} \land \phi_p^{l+1}.
\end{align*}
$$

Also note that the model of this formula, the satisfying evaluation of input variables, is a counterexample trace (one of them to be precise) which the SMT solver can generate while deciding the satisfiability.

Finally, to complete the verification process, one needs to be able to decide if two states are the same. Let $\rho$ and $\rho'$ be two states, of which it needs to be assessed whether they are identical, i.e. if the sets of possible evaluations of the delay variables are the same. Then the function $\delta^l_j$ represents the possible values of $d_j$ in $\rho$, provided that $|= pc(\rho) \land pc(\rho')$, i.e. both path conditions are satisfiable. We propose to distinguish $\rho$ and $\rho'$ using $\psi(\rho, \rho') := \Psi_{\rho, \rho'} \land \Psi_{\rho'}$ – a formula which is satisfiable iff $\rho$ and $\rho'$ are two different states:

$$
\begin{align*}
\Psi_{\rho, \rho'} : pc(\rho) \land (\bigwedge \psi_{\rho, \rho'}(i_j)) (pc(\rho')(i_j,y_{a+b}) \Rightarrow \bigwedge_{j=1}^{l-1} i_j^{l-1} \neq i_j^{l-1}(i_j/a,y_{a+b})).
\end{align*}
$$

Put together the SAT-based verification of Simulink works as shown in Algorithm 1 – we only describe generation of one successor of a state $s$, the rest is made the same as in standard model checking. That is, a particular accepting cycle detection algorithm traverses the state space by the means of successor generation. Given that the states already visited are known, a property that has to be preserved by any implementation of successor generation, the model checking process is indeed similar to the standard explicit model checking, described for example in [25].

The functions in teletype font have their expected meaning, i.e. $\text{sym}$ and $\text{exp}$ return the symbolic and explicit part of the state given as their parameter. As described in Section II, the symbolic part is the set of evaluations of input variables (here succinctly represented as the sequence of specification formulae indices) and the explicit part is the control state (here only the state of the Büchi automaton is relevant). One interesting observation is that the search for equivalent states (loop on lines [15][10] is in standard model checking a constant time operation, using hashing function. But our approach prohibits hashing because a representation using BV function is not canonical, leading to identical states with different hashes.

The correctness of the proposed approach can effectively be reduced to the correctness of $\psi$ with respect to the semantics of Simulink, i.e. $\forall \rho, \rho' : \psi(\rho, \rho') \Leftrightarrow \sigma_\rho \neq \sigma_{\rho'}$. This can then be reduced to showing two inclusion (one is $\Psi_{\rho, \rho'} \Leftrightarrow \sigma_\rho \subseteq \sigma_{\rho'}$). However, the details of the proof are rather technical and lie outside the scope of this paper.

IV. RESULTS

We report the results of the verification of the VoterCore diagram, which was partly presented in Section II. the figure shows one third of the whole diagram, the other two thirds are omitted via the $c_3$ constant. The diagram was verified against three LTL specifications: $\phi_1 = G(s_{d23} > 1 \land \neg s_{v1}) \Rightarrow s_{m1}$. $\phi_2 = G(p_{m1} \Rightarrow \neg p_{m1})$, and the last one was $\phi_3 = G(s_{m1} \land Xs_{m1} \land Xs_{m1} \land XXs_{m1} \Rightarrow F p_{m1})$, where the model satisfied $\phi_1,\phi_2$ and was incorrect with respect to $\phi_3$. The previous, purely explicit solution, described in [3] and referred to as expl, is compared with the two solutions proposed in this paper: verification using explicit sets (set) and the SAT-based verification (sat).

A. Implementation

Both explicit-set and SAT-based approaches to verification of Simulink diagrams were implemented in the DiVinE [11] verification environment, which already supported these diagrams as input, in the form of CESMI intermediate language. The implementation of sat is a relatively straightforward extension of expl: instead of keeping the evaluations in separate states.

Algorithm 1: Successor generation

<table>
<thead>
<tr>
<th>Input</th>
<th>storage of seen states $S$; state $s$; specification automaton $A_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>$S'$ where all successors of $s$ have been correctly added</td>
</tr>
</tbody>
</table>

1. foreach $\varphi_i, b'$ such that $exp(s) \models b' \in A_0$
2. $\rho \leftarrow \text{sym}(s), i$
3. if SAT_QBV(pc($\rho$)) then
4. $\text{seen} \leftarrow \text{false}$
5. foreach $s' \in S$ such that $exp(s') = b'$ do
6. $\rho' \leftarrow \text{sym}(s')$
7. if SAT_QBVw($\rho, \rho'$) then
8. $s'$ is a successor of $s$
9. $\text{seen} \leftarrow \text{true}$
10. break
11. if $\text{seen}$ then
12. $exp(s'), \text{sym}(s'') \leftarrow b', \rho$
13. $S' \leftarrow \text{store}(S, s'')$
14. $s''$ is a successor of $s$

Code available at \url{http://anna.fi.muni.cz/~xbauch/code.html#simulink}
there are fewer multi-states for every explicit control state. Furthermore, only the delay variables are represented; the import variables are stored separately only once for the whole diagram.

The implementation of sat first needed the expressions represented by Simulink blocks to be translated into BV formulae. Once a syntactic substitution was implemented is was easy to support the model checking process with a function that generates successors, as can be observed in Algorithm 1. Initially, the state matching was implemented exactly as described, where only the import variables were quantified. But the experiments revealed that much better verification times can be achieved when all variables are quantified and the substitution is not recursive, but uses only the previous history version of a particular variable. At least for the Z3 solver, this leads to better results. Communication with the SMT solver is facilitated by the SMT2 format, and thus any solver of quantified BV theory could have been used; to the best of our knowledge, however, Z3 is the only one at this time.

As mentioned in Section II, the states comprise of two parts: explicit and symbolic. The explicit part contains, most importantly, the state of the Büchi automaton and the model checker uses hashing on this part of a state to partially solve the state matching. The symbolic part contains the path condition vector and must be excluded from hashing. During successor generation, the path condition of the original state (in a conjunction with a formula labelling the Büchi automaton transition) is tested for satisfiability. The resulting formula does not contain quantifiers, and thus a decision procedure for quantifier-free BV is sufficient: line 3 of Algorithm 1.

B. Experiments

Similar execution conditions were chosen for the comparison between the purely explicit approach (unmodified DiVinE) and the new hybrid approaches: the codes were compiled with optimisation option -O2 using GCC version 4.7.2 and ran on a dedicated Linux workstation with quad core Intel Xeon 5130 @ 2GHz and 16GB RAM. Given that DiVinE uses on-the-fly verification – the traversal terminates when an accepting cycle is detected – the progression of the verification process is prone to vary. Furthermore, parallel algorithms tend to differ in the runtime between individual executions, hence we have executed every experiment 10 times and report the average of those.

1) Purely Explicit versus Explicit Sets: The 6 plots in Figure 1 contain all the measured experiments: the top row reports time results and the bottom row reports space results; each column refers to experiments against different temporal specification \( \phi_1, \phi_2, \text{ and } \phi_3 \) described above. Each plot has the range of import variables placed on the linear x-axis and either time or space on the logarithmic y-axis. The crucial observation to be made from the figures is that the Set experiments scale with the range of variables much more effectively than the Exp experiments. This observation, however, is obfuscated by two phenomena: the difference between parallel and sequential algorithms (treated in the next paragraph) and the results for property \( \phi_3 \). The VoterCore system is not a model of \( \phi_3 \) and hence there is an accepting cycle in the system. On-the-fly verification enables very fast detection in some cases (when the cycle is near the initial state) and thus the Exp experiments are better on smaller ranges (though the time and space are negligible for both approaches), because for Set experiments the generation of every successor involves nontrivial amount of computation.

The difference between parallel and sequential algorithms used in the experiments deserves placing under closer scrutiny. Parallel algorithms (Exp-par and Set-par in the plots) employ breadth-first search unlike sequential algorithms (Exp-seq and Set-seq) that employ depth-first search. Given that the resulting transition system is very shallow (the longest paths have approx. 20 edges) and at the same time extremely wide (the data-flow explosion causes each state to have many successors), the depth-first approach has a considerable advantage with respect to the space complexity. This can be observed on all the right-hand side figures for Exp experiments, yet the Set experiments cannot profit from the same phenomenon: the data-flow explosion of the number of successors was transformed into the complexity of their generation, which is common to both parallel and sequential approaches. Hence while retaining the parallel scalability (Par experiments use all four processor cores) in time, in space the Set experiments demonstrate superiority only compared to Par experiments. (\( \phi_1 \) experiments are anomalous because the property transition system has no branching and thus Set experiments traverse a linear system and cannot scale in parallel even in theory.)

2) Explicit Sets versus SAT-Based Verification: The next 6 plots in Figure 2 compare the two approaches proposed in this paper. The setting is exactly the same as before, except this time we do not include the results of parallel verification not to obfuscate the main message. Most importantly, for the VoterCore diagram and for domains of input variables larger than approximately 40, the SAT-based representation surpasses the one using explicit sets. The plots only report a part of the experiments, the limit on the x-axis is artificial so that the phenomena before the range of 40 remain clear. In fact the Sat experiments continued to a range corresponding to the domain \([0, 2^{32})\) of standard unsigned integers, which degraded the time performance at most twice compared to, for example, the range 10. Furthermore, the spacial requirements of such verification were almost negligible for arbitrary range, rarely higher than 500 MB.

For very small ranges of variables, strictly smaller than 40, the spacial requirements of the Set approach are tolerable and the time complexity is better than for the Sat approach. The outstanding spikes in the complexity of Sat also deserve further explanation. They correspond to ranges \(2^n - 1\) for smaller \(n\), i.e. the phenomenon disappears for \(n\) larger than 4. We postulate that this relates to the specific procedure that Z3 uses to decide satisfiability of quantified BV formulae. The precise explanation would probably require a detailed exposition of the functioning of the particular decision procedure employed.
in Z3, and we encourage the interested readers to follow it in [26]. The high level process in fact incorporates a model checking of quantifier instantiations, guided by previously tried unsuccessful counterexamples. It appears that limiting the range to a value very close to $2^n$ forces a larger number of wrong instantiations, but as $n$ increases the solver is able to find a good instantiation faster.

3) Discussion: To summarise the findings of our experiments with verification of Simulink diagrams, let us propose a combination of the set and sat approaches. Such a hybrid verification is possible, because the two approaches allow identical models as the input. The result would correspond to the composition of (the better of) the solid and dotted lines of Figure 2. The developer could decide which representation to use, based on the range of variables they need to verify against. Or – better yet, and fully automatically – the two approaches can be run in parallel, the one finishing first reporting the results of the verification.

V. ACKNOWLEDGMENT

This work was supported by the Czech Grant Agency grant No. GAP202/11/0312.

VI. CONCLUSION

This paper combines explicit model checking with a representation of data variables based on sets of possible evaluations. The resulting verification method allows temporal verification of systems that read from nondeterministic inputs. When the sets of evaluations are represented symbolically – as bit-vector formulae – the major obstacle that we faced and solved is the solution of state matching. The state matching procedure proposed here requires deciding satisfiability of quantified bit-vector formulae, which is a resource-intensive process, but the experimental evaluation on Simulink diagrams indicates that for such systems is the cost acceptable. For the use in verification of parallel programs or at least safety critical segments of programs, the cost would probably be prohibitive. We discuss ideas that could lead to expressing the state matching problem without the use of quantifiers, but leave that direction of research to future work.

In the case of Simulink diagrams, our approach scales with the ranges of input variables and thus allows the developer to use the range limit of the used data type, e.g. 32 bits integers, where before the developers had to test for ranges that were exponentially smaller. This improvement effectively automates the verification process, replacing completely any manual testing, since the resulting confidence in the correctness of the system with model checking is at least as high as it was with classical testing.

REFERENCES

Fig. 2: Plots that report the results of VoterCore Simulink diagram verification, comparing the representations based on explicit sets (set) and $BV$ formulae ($sat$). The range $z$ of input variables on the $x$-axis corresponds to the domain $\{0 \ldots z\}$. The set experiments had a memory limit of 10GB.