Automatic verification of programs and computer systems with data non-determinism (e.g., reading from user input) represents a significant and well-motivated challenge. The case of parallel programs is especially difficult, because then also the control flow non-trivially complicates the verification process. We apply the techniques of explicit-state model checking to account for the control aspects of a program to be verified and use set-based reduction of the data flow, thus handling the two sources of non-determinism separately.

We build the theory of set-based reduction using first-order formulae in the bit-vector theory to encode the sets of variable evaluations representing program data. These representations are tested for emptiness and equality (state matching) during the verification and we harness modern satisfiability modulo theories solvers to implement these tests.

We design two methods of implementing the state matching, one using quantifiers and one which is quantifier-free, and we provide both analytical and experimental comparisons. Further experiments evaluate the efficiency of the set-based reduction method, showing the classical, explicit approach to fail to scale with the size of data domains. Finally, we propose and evaluate two heuristics to decrease the number of expensive satisfiability queries, together yielding a 10-fold speedup.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Formal methods, Model checking; F.4.1 [Mathematical Logic and Formal Languages]: Temporal Logic

General Terms: Verification, Automation

Additional Key Words and Phrases: model checking, static analysis, modular arithmetic

1. INTRODUCTION

Verification of correctness of computer programs is decidable only in certain restricted cases, when both the program under verification and the verified property satisfy some well-defined criteria. Thus from the theoretical point of view, even deciding termination is impossible for general programs, mathematically represented for example by Turing machines. On the other hand, execution of many real-world programs entails only finitely many states of computation. Even parallel programs with a bounded number of threads that do not use dynamic data structures generate only finite state spaces. It follows that verification of many interesting properties is feasible and the only limitation is the efficiency of handling the size of the state spaces.

In the case of verifying parallel programs that read from user input against properties specified in Linear Temporal Logic (LTL), the state space explosion stems from three sources. (1) The verification complexity is exponential in the size of the formula being verified. In this paper we assume sufficiently small formulae, so that their impact on the complexity is negligible, as is often the case in practice. (2) The program under verification is commonly very large, especially considering that the number of
possible thread interleavings is exponential in the number of parallel threads. (3) The number of possible evaluations of internal program variables is much larger still, given that the introduction of every input variable adds $2^n$ new combinations, where $n$ is the number of bits and may thus be even larger than 64.

This paper is concerned with the above described problem of verifying parallel programs with inputs against LTL properties. We argue why it is reasonable to separate the control (specification formula and thread interleaving) from data (variable evaluation) into two distinct sources of non-determinism that should be handled by different means. The means used in this paper are parallel explicit-state model checking (Büchi automata-based) and satisfiability (SAT)-based symbolic execution. The proposed verification technique introduces the symbolic representation of data into otherwise unmodified LTL model checking or, equivalently, it promotes symbolic execution to LTL verification by implementing exact state matching.

More precisely the proposed verification enumerates explicitly all reachable control configurations (combination of program counters and specification automaton states) while the set of variable evaluations associated with a control state is represented symbolically, as a formula in the first-order theory of bit-vectors ($\mathcal{BV}$). Hence, instead of storing many copies of the same control state that differ in the variable evaluation, our verification represents the states with exactly the same control part and exactly the same set of evaluations as a single multi-state. This set-based state space reduction leads in some cases to exponentially smaller state spaces and we prove the correctness of the reduction and discuss its efficiency. Also, unlike in classical symbolic model checking where states are characteristic functions of sets of configurations (one for both control and data), we represent data as images of $\mathcal{BV}$ functions.

Representing sets of variable evaluations as function images has the potential to be more concise than using a canonical representation of characteristic functions, e.g., more than Binary Decision Diagrams (BDDs), which are exponential in the number of variables for some arithmetic functions [Bryant 1991]. Furthermore, evaluating the transition relation is syntactic, a simple substitution, which allows very fast generation of state successors. On the other hand, one of the contributions of this paper is the identification of a major limitation entailed with using function images for model checking: deciding state matching (equivalence of two sets) is very expensive.

In the context of model checking, it is crucial to be able to decide when two objects in the state space represent the same state. State matching is used in accepting cycle detection and for deciding termination; both these operations are necessary for the correct functioning of the verification process. In order to utilise the advance of modern SAT and Satisfiability Modulo Theories (SMT) solvers, we have devised an SMT query for state matching: a $\mathcal{BV}$ formula is satisfiable if and only if two given functions have different images. Yet such a query requires quantifying the input variables and thus deciding its satisfiability is, in theory, exponentially harder than quantifier-free satisfiability. Our experiments with randomly generated and DVE (native language of DIVINE) models demonstrate that exact state matching limits our approach to rather small models of programs.

In an attempt at ameliorating the complexity of state matching, this paper follows two directions: using quantifier-free state matching for the price of its precision and modifying the way the model checker searches the database of already visited states. Consider first the quantifier-free state matching implemented by comparing the observable behaviour of the two functions; let us call two states different if and only if their functions map a single input to two distinct outputs. We prove that this equivalence-based state matching, while not precisely comparing two sets of evaluations, leads to a correct model-checking process which terminates provided that input variables are not read on a cycle. The second improvement decreases the number of
SAT calls related to the insertion of a new state into the model checker database. The number of symbolic data parts associated with a single explicit control part can be decreased by explicating some variables which can be efficient if there is exactly one concrete value in the image. Finally, some states can be distinguished by reusing the satisfying models gained from previous SAT calls (evaluation of input variables that witness the difference between states). We have built a hierarchy atop these witness models which in some cases allows simulating a binary search among database states. The proposed set-based state space reduction and the associated control explicit—data symbolic model checking are experimentally evaluated, together with the above described improvements, on DVE models and randomly generated test instances. The experiments compare data representations using explicit sets with those using BV formulae, precise image-based state matching with equivalence-based state matching, and we also evaluate the influence of explicating variables and the witness model hierarchy.

There is a number of interesting observations to be highlighted. For example, there is a threshold on the size of the input variables domain below which the explicit representation outperforms the symbolic due to the high complexity of exact state matching. On the other hand, verification with the symbolic representation scales gracefully with the increase of data domains (32-bit domains caused less than two-fold slowdown compared to 4-bit domains). Regarding the two state matching techniques, the image-based comparison is two orders of magnitude slower than the quantifier-free equivalence-based comparison in a vast majority of cases. Yet our experiments with DVE models lead to relatively short quantifier-free SMT queries that required up to 88 seconds to be resolved by state-of-the-art SMT solvers. Finally, the witness-based matching reduces the number of SMT solver calls exponentially in some cases, yet all unsatisfiable instances remain (these are more expensive than the satisfiable ones) and thus the resulting impact is only a linear, 7-fold speedup.

Contribution. This paper summarises our work on bit-precise LTL model checking of parallel programs with inputs. The Simulink diagram verification presented in [Barnat et al. 2014b] also included the methodology of translating state matching into quantified SMT queries. The comparison between image-based and equivalence-based state matching and the explicating values heuristics were presented in [Barnat et al. 2014a]. This publication not only considerably extends both above-mentioned papers, with further optimisations and experiments, but more importantly it establishes the theory of set-based verification, which the work described before is an instance of. We provide detailed proofs of correctness of set-based verification and the correctness of both state matching techniques and the relation between them. The previously unpublished witness-based matching resolution heuristics is evaluated on randomly generated models, which are also used to gain deeper insight into the relation between the two state matching techniques. Finally, Section 3.5 applies all the accumulated theory in a comprehensive description of LTL model checking built on the set-based reduction.

The novelty of the verification methodology proposed here lies in the combination of explicit state model checking with sets of variable evaluations represented as images of bit-vector functions. We build the theory of set-based reduction using the above combination for verification of systems with asynchronous control-flow and regular, arithmetical data-flow, e.g., for parallel programs with input variables. The result of incorporating the set-based reduction into an existing explicit-state model checker DI- VINE [Barnat et al. 2013] effectively allows verification of programs reading from input, which was previously impossible in all non-trivial cases. Translating these results to software development, we have extended the limits of previous complete, bit-precise techniques to the point when the system designer no longer has to bound the domains
of input variables. Instead of iteratively testing the model on domains of sizes smaller than 100, the set-based reduction and symbolic representation allow verifying for the whole domain $0 - 2^{32}$ in a single verification run.

2. PRELIMINARIES

Following [Manna and Pnueli 1995] we assume that first-order (FO) formulae refer to variables from a universal set $V$. Each variable $x \in V$ draws its values from a domain $\text{sort}(x)$. The set of evaluations $\mathcal{E}$ of variables from $V$ contains functions from $V$ to $\bigcup_{x \in V} \text{sort}(x)$, such that $\forall (\nu \in \mathcal{E}), \nu(x) \in \text{sort}(x)$. Let $X \subseteq V$ denote a set of variables $\{x_1, \ldots, x_n\}$, we assume a fixed ordering and overload the notation to allow $\nu(X) = (\nu(x_1), \ldots, \nu(x_n))$.

In order to model precisely the semantics of programs, the domain of program variables consists of fixed-width bit vectors, i.e., $\text{sort}(x) = \{0, \ldots, 2^{q_x} - 1\}$ for some $q_x$. The variable $q_x$ denotes the bit width of $x$, thus for example int32_t $x$; declares $x$ to be a 32-bit integer variable with $q_x = 32$ (using C notation). Apart from program variables, $V$ also contains program counters $pc$, one for each parallel process, where $\text{sort}(pc)$ is the set of program locations of a given process. Finally, a subset $I \subseteq V$ denotes an infinite set of input variables, drawing non-deterministic values from $\text{sort}(\iota) = \{0, \ldots, 2^{q_\iota} - 1\}$ for $\iota \in I$. Altogether, $V = D \cup D' \cup \{pc, pc'\} \cup I$, where $D$ is the set of data variables (the next-state versions are primed), $pc$ the program counter, and $I$ the set of input variables. Then the state of a program is uniquely defined as an evaluation of the control and data variables from $D \cup \{pc\}$.

The FO theory used in this paper is the bit-vector $BV$ theory with modular Peano arithmetic and bit-wise operations that when applied to variables from $V$ form bit-vector expressions [Kroening and Strichman 2010]. We denote both functions and predicates with Greek letters and the set of all $BV$ expressions over $V$ as $\Phi$.

For modelling parallel programs, their processes, and specification automata we use the flow graphs.

**Definition 2.1.** The control flow graph (CFG) is a tuple $P = (L, l_0, T, F)$, where

- $L$ is the set of program locations
- $l_0 \in L$ is the initial location
- $T \subseteq L \times \Phi \times L$ is the transition relation
- $F \subseteq L$ is the set of final locations

and all $L$, $T$, and $F$ are finite.

Hence a transition $t \in T$ is a triple $t = (l, \rho, l')$, $l, l' \in L$, where $\rho$ describes the relation between current state and next state data variables; depicted visually, $t = l \xrightarrow{\rho} l'$. Example 2.1 shows CFGs of parallel processes of a program, where the parallel composition operator $\parallel$ will be defined in Definition 2.2.

**Example 2.1.** Let us demonstrate how threads of a parallel program can be mapped to our definition of CFGs. The parallel program on the left is represented by the three CFGs on the right (the top one only initialises the global variables with non-deterministic values). The transition labels are simplified in that each variable preserves its value, i.e. $x' = x$, whenever it is not explicitly modified.
For example the CFG of thread 1 is $G = (L, l_1, T, \emptyset)$, where

- $L = \{ l_1, l_2, \hat{l}_1 \}$
- $T = \{ (l_1, a > 5 \land a' = a \land b' = b, l_2), (l_1, a \leq 5 \land a' = a \land b' = b, \hat{l}_1), (l_2, a' = a \land b' = b + 1, \hat{l}_1) \}$

\[ g_1 : \text{uint32}, t a, b; \hat{g}_1 \]
\[ \text{thread1}() \{ \]
\[ l_1 : \text{if} (a > 5) \{ \]
\[ l_2 : b + +; \} \hat{l}_1 \]
\[ \text{thread2}() \{ \]
\[ m_1 : \text{while} (b < 20) \{ \]
\[ m_2 : a = (a + 13) \% 5; \} \hat{m}_1 \]

It will prove useful in future discussion to introduce three composition operations on CFGs: (1) the parallel $\parallel$ to model interleaving of parallel processes, (2) the synchronous $\triangle$ to model the connection between a program and its specification, and (3) the sequential $;$ to model the sequential transfer of control from one CFG to the next. To simplify the notation we define compositions between two CFGs, but the extension to more CFGs is straightforward.

**Definition 2.2.** The parallel composition $P = P_1 \parallel P_2$ of CFGs $P_1 = (L^1, l_0^1, T^1, F^1)$ and $P_2 = (L^2, l_0^2, T^2, F^2)$ is a CFG $P = (L, l_0, T, F)$, where

- $L = L^1 \times L^2$
- $l_0 = (l_0^1, l_0^2)$
- $T = \{(l_1, l_2) \xrightarrow{\rho} (l_1', l_2') \mid (l_1 \xrightarrow{\rho} l_1' \in T^1 \land l_2 = l_2') \lor (l_2 \xrightarrow{\rho} l_2' \in T^2 \land l_1 = l_1') \}$
- $F = \{(t_1, t_2) \mid t_1 \in F^2 \lor t_2 \in F^2 \}$

**Definition 2.3.** The synchronous composition $P = P_1 \triangle P_2$ of CFGs $P_1 = (L^1, l_0^1, T^1, F^1)$ and $P_2 = (L^2, l_0^2, T^2, F^2)$ is a CFG $P = (L, l_0, T, F)$, where $L$, $l_0$, and $F$ are defined similarly as for the parallel composition and

- $T = \{(l_1, l_2) \xrightarrow{\rho} (l_1', l_2') \mid l_1 \xrightarrow{\rho_1} l_1' \in T^1, l_2 \xrightarrow{\rho_2} l_2' \in T^2, \rho = \rho_1 \land \rho_2 \}$

**Definition 2.4.** The sequential composition $P = P_1 ; P_2$ of CFGs $P_1 = (L^1, l_0^1, T^1, F^1)$ and $P_2 = (L^2, l_0^2, T^2, F^2)$ is a CFG $P = (L, l_0, T, F)$, where we assume $L^1$ and $L^2$ to be distinct and

- $L = L^1 \cup L^2$
- $l_0 = l_0^1$
- $T = T^1 \cup T^2 \cup \{(l_1 \xrightarrow{\rho_1} l_0^1) \mid l_1 \in F^1 \}$
- $F = F^2$

**Example 2.2.** Let $P_1$ and $P_2$ be the CFGs of threads 1 and 2 from Example 2.1 and $P_3$ the CFG of the global process. Below is the composition $P = P_3; (P_1 \parallel P_2)$, where some of the transition labels were omitted to avoid cluttering.
The CFGs represent programs in a control explicit, data symbolic manner. Explicit-state model checking, however, operates over fully explicit systems. In order to define model checking of programs we must first explicate the manipulation of data. A standard formalism for the representation of models of programs is the labelled transition system: effectively a CFG where each transition is labelled with a true formula. This means that all the information about the variable values has to be stored in the control-flow locations. Since a large part of our work is concerned with reducing the size of the resulting CFG, we represent each reduction technique as a process of generating the explicit CFG.

**DEFINITION 2.5.** The explicit control-flow graph is a CFG $G = (S, s_0, T, A)$, where for each $s \xrightarrow{\sigma} s' \in T$ it holds that $\sigma = \top$ and we simplify the notation to $s \rightarrow s'$. From a CFG $P = (L, l_0, T, F)$ we can generate the explicit CFG $G$. The set of locations $S$ is the set of evaluations of $\{pc\} \cup D$. We define the generation as follows:

- $S = \{ s : \{pc\} \cup D \rightarrow L \cup \bigcup_{x \in D} \text{sort}(x) \mid s(pc) \in L, \forall x \in D, s(x) \in \text{sort}(x) \}$
- $s_0 = \{ (pc, l_0) \} \cup \{ (x, 0) \mid x \in D \}$
- $T = \{ s \rightarrow s' \mid s(pc) \not\rightarrow s'(pc) \in T, s = \{s(x) \land \rho \land \land x' = s'(x) \} \}$
- $A = \{ s \in S \mid s(pc) \in F \}$

The above definition of transitions in the explicit CFG requires that the transition relation $\rho$ holds between the evaluation in $s$ and the evaluation in $s'$. Note that $\rho$ may also refer to the input variables from $I$. Hence the fact that the formula is satisfiable, denoted by $\models_{SAT}$, states that there is an evaluation of the input variables for which the formula is valid.

### 2.1. Verification of LTL Properties

As stated in the introduction, LTL is a logic that allows developers to specify temporal properties of their programs. The logic is based on so-called atomic propositions, i.e., formulae from $\Phi$ which can be evaluated over locations of the explicit CFG.
**Definition 2.6.** For an atomic proposition $p \in \Phi$, we define the syntax of Linear Temporal Logic inductively as follows:

$$\psi ::= p \mid \neg \psi \mid \psi \land \psi \mid X \psi \mid \psi U \psi$$

Intuitively, $X \psi$ states that $\psi$ will hold in the next program state and $\psi_1 U \psi_2$ that $\psi_1$ will hold up to some point in the future at which $\psi_2$ holds. The full and precise semantics of LTL relates to the specific transition system which must be defined first.

Let us add that there are other linear operators that can be defined using $X$ and $U$, e.g., $F \psi := \top U \psi$ or $G \psi := \neg F \neg \psi$ formalise the intuitive notion of formulae that hold at some point in the future or globally (in every program state), respectively.

Finally, LTL formulae describe infinite behaviour and thus we use infinite sequences for defining the semantics of LTL. Formally, an infinite sequence of locations from $L$, $w = (s_0, s_1, \ldots)$, is a mapping $w : \mathbb{N} \rightarrow L$.

**Definition 2.7.** Let $G = (S, s_0, T, A)$ be an explicit CFG and $\psi$ an LTL formula. A run (or trace) $w$ of $G$ is an infinite sequence of explicit locations $w = (s_0, s_1, \ldots)$ and $w_i$ its $i$-th suffix, i.e., $w_i = (s_i, s_{i+1}, \ldots)$. The satisfaction of $\psi$ in a run $w$ is defined inductively, according to the structure of $\psi$:

- $w \models p \iff s_0 \models p$
- $w \models \neg \psi \iff w \not\models \psi$
- $w \models \psi_1 \land \psi_2 \iff w \models \psi_1$ and $w \models \psi_2$
- $w \models X \psi \iff w_1 \models \psi$
- $w \models \psi_1 U \psi_2 \iff \exists (i \in \mathbb{N}), \forall (j < i), w_j \models \psi_1$ and $w_i \models \psi_2$

We overload the entailment relation $\models$ to denote (1) that a trace $w$ satisfies an LTL formula $\psi$ and (2) that an evaluation of variables from $X$ is a model of a $BV$ formula over variables from $Y \subseteq X$. Finally, the whole CFG $G$ satisfies $\psi$ if all of its runs do. The task of an LTL model checker is to provide an answer \{correct, (incorrect, counterexample)\} for an input pair $(G, \psi)$, where counterexample is a run of $G$ that does not satisfy $\psi$.

**Example 2.3.** We continue Example 2.2 with an example run $w$ of $P$. We represent a transition $(s, \rho, s')$ as $s \xrightarrow{\rho} s'$. Let us assume that the program reads $\iota_1 = 6$ and $\iota_2 = 11$ as the input values and that we are considering the satisfaction of an LTL formula $\psi : G b < 12$. The run $w$ does not satisfy $\psi$: even though $b < 12$ holds in the first seven states, the eighth state is not a model of $b < 12$ and thus even this finite prefix demonstrates that $w$ does not satisfy $\psi$. Note that $w$ is only one of many runs of the system, in fact there are exactly $2^{12}$ distinct successors of the first state of $w$. 

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, Article A, Pub. date: January YYYYY.
2.2. Explicit-State Model Checking
A classical approach to verify LTL properties is automata-based model checking (see the relevant parts of Section 5 for other approaches). Informally, for an LTL formula $\psi$ one can construct an $\omega$-automaton – an automaton over infinite words – that accepts exactly those words that model the formula $\psi$. Hence the automaton for formula $\neg \psi$ accepts exactly the counterexamples for the formula $\psi$. It follows that if a product automaton of the program $P$ and the automaton for $\neg \psi$ is constructed, then it accepts exactly those counterexamples for $\psi$ that are legal executions of $P$. The verification is thus reduced to locating such accepted words.

The $\omega$-automata used to represent LTL formulae are called Büchi automata [Clarke et al. 1999], but their structure is similar to CFGs and thus we do not introduce a new formalism. For our purposes it suffices to define the set of words accepted by a CFG.

**Definition 2.8.** A CFG $P = (L, l_0, T, F)$ accepts a trace $w = (s_0, s_1, \ldots)$ if and only if both following conditions hold

- $\forall (i \in \mathbb{N}), w(i)(pc) \not\in w(i+1)(pc) \in T \land w(i) = \rho$
- $\forall (j \in \mathbb{N}), \exists (i \in \mathbb{N}), i > j \land l_i \in F$

The first condition of the above definition requires that each location in $w$ satisfies the relevant transition in $P$. The second condition requires that the sequence of locations in $w$ visits the set of final states $F$ infinitely many times.

Henceforth, we do not need to distinguish between the original program and its CFG representation. Assuming that $P_A$ is the CFG for the LTL property being verified, we fix the notation $P = (P_1 \parallel \ldots \parallel P_n) \parallel P_A = (L, l_0, T, F)$ and only refer to $P$ as the input for model checking. Words accepted by $P$ are the witnesses of the program’s failure to meet its specification. Thus if the set of accepting words is empty then the program is correct. The existence of accepting words is equivalent to the existence of reachable accepting cycles in the explicit state space $G = (S, s_0, T, A)$. A cycle is a closed path $\zeta = s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n = s_1$ and $\zeta$ is accepting if $\exists (1 \leq i < n), s_i \in A$. A state $s_i$ is reachable if there is a path from $s_0$ to $s_i$ and, by extension, a cycle is reachable if any of its states is reachable. Thus in classical explicit-state model checking, LTL
verification is reduced to finding accepting cycles within the transition system of the program/specification product.

Example 2.4. Let us conclude the demonstration started in Example 2.1 by presenting the CFG $P_A$ (on the left) for verifying formula $\varphi : G b < 12$, i.e. representing runs that satisfy its negation: $\neg \varphi = \neg F (b \geq 12)$. The sourceless arrow points to the initial location.

Assuming that $w = (s_0, s_1, s_2, \ldots)$ is the run from Example 2.3, then the path on the right represents a reachable accepting cycle in the explicit CFG for the synchronous product of $P_A$ and the program from Example 2.2. This cycle can be mechanically detected, thus demonstrating the program’s incorrectness with respect to $\psi$. △

Without going into too many implementation details, we briefly describe the internal functioning of an explicit model checker (a more detailed exposition can be found at the end of Section 3). Assume that the input is the composition $(P_1 \parallel \ldots \parallel P_N) \parallel P_A$ but only in an implicit form: the parallel processes and the specification automaton are known but the state space of the composition has to be generated. During accepting cycle detection, the model checking algorithm generates successors of individual states using Definition 2.5. The set of visited states is maintained in an explicit storage and every newly generated state is tested for being previously visited. Based on the specific cycle detection algorithm the state space is traversed in some order, until either an accepting cycle is found or no new state can be generated.

As a final note and to motivate the state space reductions proposed in this paper, let us consider model checking of a correct system. To demonstrate correctness, the verification process has to refute the existence of accepting cycles, which, in the explicit-state case, entails enumerating the whole transition system. The part of Definition 2.5 describing successor generation allows us to properly appreciate the size of such transition systems. Let us fix a state $s_i$ of the explicit CFG $G$ and compute the number of possible successors of $s_i$. Let $P^A = (L^A, l_0^A, T^A, F^A)$ be the specification CFG and $P^i = (L^i, l_0^i, T^i, F^i)$ the CFGs all the of individual processes, where $n$ is the number of processes and $0 < i \leq n$. Given the nature of parallel and synchronous compositions, the product $P = (P^1 \parallel \ldots \parallel P^n) \parallel P_A$ has $O(L^A \cdot \Pi_{i=1}^n L^i)$ locations. Hence the number of possible successor control states is exponential in the number of parallel processes, but often reasonably small. On the other hand, the evaluation of program variables entails $O(\text{sort}(i))$, $i \in I$ successor data states each time an input is read. Note also that the state spaces of individual processes $P^i$ and of the automaton $P_A$ are usually significantly smaller than $\text{sort}(i)$. Furthermore, the number of control successors is commonly constant, i.e., independent of the number of locations. This is exactly the opposite case for variable evaluation: reading from input very often leads to $\text{sort}(i)$ successors. This observation lies at the core of the thesis mentioned in the introduction: the two sources of non-determinism in parallel programs are of different nature and should thus be handled by different means.
3. METHODS

The thesis on which this work is based can be summarised as follows: explicit-state model checking may be seamlessly extended to handle the control part of programs explicitly while substituting a symbolic representation for the data part. Not all symbolic representations are appropriate for the data part, and indeed not all forms of data are appropriate for LTL model checking. The reason is that most symbolic representations were created to support the checking of safety properties. Yet LTL model checking requires exact state comparison, while checking safety properties does not. This section introduces one possible representation of symbolic data, i.e., a state space reduction technique. This representation will later prove adequate for arguing about correctness and as a specification that will be refined in the actual implementation.

3.1. Set-Based Reduction

The proposed method of handling the data flow consists of clustering together those variable evaluations that are connected to the same control state. This way, most of the aspects of explicit state model checking can be preserved provided that one can efficiently store these sets of evaluations. More precisely, we want to propose a reduction method that would modify the state space in such a way that an unmodified model checking of this reduced space would yield exactly the same result as it would for the unreduced state space. That clearly entails the ability to efficiently decide whether a state has already been seen, using a fast storage data-structure and state matching decision procedures.

Unlike the explicit CFG which represents the unreduced states space, the set-based reduction method does not include data explicitly within states. To simplify further argumentation, we introduce a new variable data, such that sort(data) = \{(v_1, \ldots, v_n) \mid v_i \in \text{sort}(x_i)\}. Hence sort(data) is a set of vectors \(v\) and a vector element \(v_i\) is associated with the value of a variable \(x_i\) such that \(v_i \in \text{sort}(x_i)\).

**Definition 3.1.** The set-reduced CFG \(G = (S, s_0, T, A)\), similar to the explicit CFG, limits the transition labels to \(\top\), but, unlike in the explicit CFG, \(S\) contains the evaluations of only two variables \(pc\) and \(data\). The generation from a CFG \(P = (L, l_0, T, F)\) is defined as follows:

\[
S = \{s : \{pc, data\} \to L \cup \text{sort(data)} \mid s(pc) \in L, s(data) \in \text{sort(data)}\}
\]

\[
s_0(pc) = l_0, s_0(data) = \top
\]

\[T = \{s \to s' \mid s(pc) \not\in s'\} \}
\]

\[\exists v_i \in s(data), |s(\top) \land x_i = v_i \land \rho, s'(data) = \{v_i' \in \text{sort(data)} \mid \exists v_i \in s(data)\}
\]

\[A = \{s \in S \mid s(pc) \in F\}
\]

Even though the representation of the evaluations of \(data\) is very important, let us, for now, abstract from it and investigate the reduction assuming that there is an efficient representation. More precisely let us assume that we have a representation whose size is linear in the number of program variables but that does not depend on the domains of these variables or on the number of times the input was read from, the length of the path from \(s_0\), etc. None of these assumptions will prove achievable using the currently known symbolic representations, but for the purpose of evaluating the general idea, we will assume them regardless.

One might observe that it is no longer obvious when two states of the reduced transition system should be considered equivalent: which needs to be decidable for the
accepting cycle detection. First, let us demonstrate that equality based on the sub-
set relation – a newly discovered state $s'$ being equal to a known $s$ provided that
$s'(data) \subseteq s(data)$ – is not correct with respect to LTL properties. In the context of
symbolic execution, this type of equivalence is known as subsumption [Xie et al. 2005],
where its use is justified by the fact that symbolic execution verifies reachability prop-
erties for which the subsumption is correct.

Example 3.1. The incorrectness of subsumption-based reduction for LTL verifica-
tion can be easily demonstrated. Consider the verification task below, with the program
CFG on the left and the specification CFG on the right.

After taking the transition $\tau_0$ the set-reduced CFG reaches a state $s$ which evalu-
ates $data$ to $\{0, \ldots, 10\}$. Taking $\tau_1$ leads to a state $s'$ with $s'(data) = \{5, \ldots, 9\}$. Under
subsumption the states $s$ and $s'$ would be equal. Given the specification CFG on the
right, each state is accepting. Under subsumption we would thus report an accepting
cycle, which would be incorrect. The incorrectness can be demonstrated by listing all
paths of the explicit CFG. In short, the $\tau_1$-successors of states where $x < 5$ evaluate $x$
to a value larger than 5 and states with $x \geq 5$ have no successors. Hence there are no
cycles in the explicit CFG. △

It appears that unless the underlying model-checking process itself is modified, the
reduction has to differentiate between any two states that have different sets of pos-
sible variable evaluations. Henceforth this reduction is referred to as set-based reduc-
tion, and for an unreduced state space $G^u$ the set-reduced state space is denoted by $G^s$
(whenever we need to distinguish it from other forms of reduction). The reduction is
demonstrated in the following example and formalised in the following subsection.

Example 3.2. Let us consider verifying the following parallel program $P$:

with respect to a specification represented by the automaton $P_A$:

requiring that at some point, the execution sets $b$ to a value larger than 10, and after-
wards never decreases it below 11.
The program $P$ is not represented as a composition of transition systems, but as a C-like pseudo-code to further demonstrate the possibility to translate real-world verification problems to the presented formalism. Below is the transition system generated without our set-based reduction:

The identification of program states can be divided into two parts: one for control information (marked with lighter blue in the figure) and the other for data (marked with darker red). Note also that the control part contains the program counters for individual threads of the main program $\bar{p}$ and the locations $p_{c_A}$ of the specification automaton $P_A$. Similarly, it is possible to distinguish the two sources of non-determinism in parallel programs: the control-flow non-determinism (thread interleaving) is marked as $\zeta$-transitions and the data-flow non-determinism (variable evaluation) as $\delta$-transitions.

Finally, the set-reduced transition system may be visualised as follows:

In the above figure we abbreviate data to $d$. Note especially how the set-reduction contracts the data non-determinism $\delta$, while preserving the distinctions prescribed by the control.

Subset inclusion was shown to lead to unification of states that does not preserve the information on the progress towards the LTL specification. In fact, of any unification among states it must be proven that if an accepting cycle was created then there is one in the unreduced system as well. This holds even for a more restrictive form of state equivalence, when two states are equal only if the evaluations they contain match exactly. The following theorem proves correctness of the unification employed in our reduction. Notice that even though atomic propositions may evaluate differently under $s(data)$ within a single state, they are divided on-the-fly, based on the labels of the specification CFG transitions. That is ensured by the way transitions are generated in Definition 3.1.

**Theorem 3.2 (Correctness).** Let $P = (L, i_0, T, F)$ be the CFG of the verified program and let $G^u = (S^u, s^u_0, T^u, A^u)$ and $G^s = (S^s, s^s_0, T^s, A^s)$ be the generated explicit and set-reduced CFGs, respectively. Then $G^s$ contains a reachable accepting cycle if and only if $G^u$ contains a reachable accepting cycle.
Lemma 3.3. For any path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$ in $G^u$ there is a path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$ in $G^u$ such that $(v(0 \leq i \leq n), s_i^u(\text{pc}) = s_i^u(\text{pc}) \land s_i^u(D) \in s_i^u(\text{data}))$.

Proof. For $n = 0$ we have that $s_0^u(\text{pc}) = i_0 = s_0^u(\text{pc})$ and $s_0^u(D) = \overline{0} = s_0^u(\text{data})$. Let us assume that the statement holds for paths of length $n$ and take an arbitrary path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u \rightarrow s_{n+1}^u$. The induction hypothesis gives us the appropriate path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$. According to Definition 2.5, the existence of $s_n^u \rightarrow s_{n+1}^u$ entails $s_n^u(\text{pc}) \rightarrow s_{n+1}^u(\text{pc}) \in \mathcal{T}$. Thus also $s_n^u(\text{pc}) \rightarrow s_{n+1}^u(\text{pc}) \in \mathcal{T}$ and since $s_n^u(D) \in s_n^u(\text{data})$ we also have $s_n^u(\text{data}) \neq \emptyset$. Already it holds that $s_n^u \rightarrow s_{n+1}^u \in T^u$ and from $\models_{\text{SAT}} x = s_n^u(x) \land \rho \land x' = s_{n+1}^u(x)$ we conclude that $s_{n+1}^u(D) \in s_{n+1}^u(\text{data})$. □

Lemma 3.4. For any path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$ in $G^u$ and for all $\tau \in s_n^u(\text{data})$ there is a path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$ in $G^u$ such that $s_n^u(D) = \tau$.

Proof. Since $s_n^u(\text{data}) = \{\overline{0}\}$ the statement holds trivially for $n = 0$. Let us assume a set-reduced path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u \rightarrow s_{n+1}^u$ and let us fix a data vector $\tau' \in s_{n+1}^u(\text{data})$. The transition $s_n^u \rightarrow s_{n+1}^u$ gives us $s_n^u(\text{pc}) \rightarrow s_{n+1}^u(\text{pc}) \in \mathcal{T}$ and $s_n^u(\text{data}) \neq \emptyset$. Also since $\tau' \in s_{n+1}^u(\text{data})$ we have that $\models_{\text{SAT}} x = s_n^u(x) \land \rho \land x' = s_{n+1}^u(x)$ for some $\tau \in s_n^u(\text{data})$. Applying the induction hypothesis, we get a path $s_0^u \rightarrow s_1^u \rightarrow \ldots \rightarrow s_n^u$ with $\tau = s_n^u(D)$ and a transition in $G^u$ from $s_n^u$ to $s_{n+1}^u$ with $s_{n+1}^u(D) = \tau'$. □

Lemma 3.5. Let $\rho \in \Phi$ be a BV formula. We can associate with $\rho$ a relation $r$ as follows:

$$r = \{((\tau, \tau') | v_1, v_1' \in \text{sort}(x_i), \models_{\text{SAT}} x_1 = v_1 \land \rho \land x_1' = v_1')$$

For each $\rho \in \Phi$ there are $m, n \in \mathbb{N}_0$ such that for all subsets $X$ of $\{\tau | v_1 \in \text{sort}(x_i)\}$ it holds that $\{r^m | \tau \in X\} = \{r^{m+n} | \tau \in X\}$.

Proof. Let $R^i = \{r^i | \tau \in X\}$ and consider the infinite sequence $R^1, R^2, \ldots$. Given that each sort$(x_i), x \in D$ is finite there are $k, l \in \mathbb{N}_0$ such that $R^k = R^l$. Assume $k > l$, then we can take $l$ for $m$ and $k - l$ for $n$ to establish the statement. □

Definition 3.6. For a collection of relations $\rho_1, \ldots, \rho_d$ from $\Theta$ we define their composition $\rho = \rho_d \circ \ldots \circ \rho_1$ as a formula

$$\rho = \rho_d[x \mapsto x^d] \land \rho_{d-1}[x \mapsto x^{d-1}, x' \mapsto x^d] \land \ldots \land \rho_2[x \mapsto x^2, x' \mapsto x^3] \land \rho_1[x' \mapsto x^2]$$

Lemma 3.7. If $s_1^u \rightarrow \ldots \rightarrow s_{c+1}^u$ is a cycle in $G^u$ then there is a path $s_{11}^u \rightarrow \ldots \rightarrow s_{c+1}^u$ in $G^u$ such that

1. $\forall (1 \leq i \leq c) (j \geq 1), s_j^u(\text{pc}) = s_j^u(\text{pc})$
2. $\exists (m, n \in \mathbb{N}), s_m^u(\text{data}) = s_n^u(\text{data})$

Proof. Unrolling the cycle $s_1^u \rightarrow \ldots \rightarrow s_n^u$ and applying the Lemma 3.3 we get a path in $G^u$ satisfying (1). Let $\rho_1$ be the formulae labelling the corresponding path in $P$, i.e., $s_1^u(\text{pc}) \rightarrow s_{n+1}^u(\text{pc})$, and let $\rho$ be the formula for the composition of $\rho_1, \ldots, \rho_c$. Applying Lemma 3.5 on $\rho$ we get the $m$ and $n$ satisfying (2). □

Lemma 3.8. If $s_1^u \rightarrow \ldots \rightarrow s_c^u$ is a cycle in $G^u$ then there is a path $s_{11}^u \rightarrow \ldots \rightarrow s_{c+1}^u$ in $G^u$ such that

1. $\forall (1 \leq i \leq c) (j \geq 1), s_j^u(\text{pc}) = s_j^u(\text{pc})$
2. $\exists (m, n \in \mathbb{N}), s_m^u(D) = s_n^u(D)$

Proof. Unrolling the cycle $s_1^u \rightarrow \ldots \rightarrow s_n^u$ and applying the Lemma 3.4 we get a path in $G^u$ satisfying (1). Let $k = |s_1^u(\text{data})|$ and consider the prefix $s_{11}^{(k+1)}$.
For this prefix it holds that \( \forall (1 \leq i \leq k + 1), s^u_{i+1}(D) \in s^u_i(data) \) and thus at least one data-vector must repeat, i.e., \( \exists (1 \leq m < n \leq k + 1), s^u_{i+m}(D) = s^u_n(D) \), as required by (2).

Proof of Theorem 3.2. Assume that \( s^u_1 \rightarrow \ldots \rightarrow s^u_m = s^u_1 \) is an accepting cycle in \( G^u \) reachable from \( s^u_0 \). Then Lemma 3.7 gives us a lasso-shaped path \( s^u_1 \rightarrow \ldots \rightarrow s^u_m \rightarrow \ldots \rightarrow s^u_m+s^u_n = s^u_n \) in \( G^u \) which also leads to a cycle. This cycle is also accepting given (1) of Lemma 3.7 and the definition of \( A^u \). Finally, applying Lemma 3.3 on the path from \( s^u_0 \) to \( s^u_1 \) proves the reachability of an accepting cycle in \( G^u \).

Assume that \( s^u_1 \rightarrow \ldots \rightarrow s^u_m = s^u_1 \) is an accepting cycle in \( G^u \) reachable from \( s^u_0 \). Then Lemma 3.8 gives us a lasso-shaped path \( s^u_1 \rightarrow \ldots \rightarrow s^u_n \rightarrow \ldots \rightarrow s^u_n = s^u_n \) in \( G^u \) which also leads to a cycle. This cycle is also accepting given the definition of \( A^u \) and reachable from \( s^u_m \) (after applying Lemma 3.4 to \( s^u_m \rightarrow \ldots \rightarrow s^u_1 \)).

Theorem 3.2 gives us the assurance that an unmodified model checker can be used together with the set-reduced transition system and would provide the correct result. Yet reasoning about the efficiency of the proposed set-based reduction – the ratio between the size of the original system and the size of the reduced system – is rather complicated, as will be shown in Example 3.3. For a program without cycles, the reduction is exponential with respect to the number of input variables and to the sizes of their domains. Note, however, that for trivial cases of data-flow non-determinism even this reduction can be negligible. The case of programs with cycles is considerably more involved.

Let us call control cycles those paths in a transition system that start and end in two states with the same control part, \( s, s' \) such that \( s(pc) = s'(pc) \). Then the relation \( r \) of the cycle, transforming \( s(D) \) to \( s'(D) \), has a fixed point as was argued in the above proof, and this fixed point has to be computed (explicitly in our case, as opposed to the symbolic solution [Lin 1996] of the same problem). That aspect is present in full and reduced state spaces alike, yet may produce an exponential difference in their sizes. If the multi-state already contains the fixed point before the generation reaches the given cycle, as in the second program of Example 3.3, then the reduced system contains only as many multi-states as is the length of that cycle. On the other hand, as the first program of Example 3.3 demonstrates, the reduction can be to the detriment of the space complexity even if we assume that the size of multi-states is sub-linear in the number of states contained within.

Example 3.3. There are two programs that nicely exemplify some of the properties of the set-based reduction, which we will use in further discussion, especially regarding the efficiency of this approach. Consider the following program with a loop:

\[
\text{read } a; \text{ while ( a > 10 ) } a--; \\
\]

When only the data part of multi-states is considered, the reduced transition system unfolds to

\[
\begin{align*}
\text{a = \{0\}} & \rightarrow \text{a = \{0..255\}} & \rightarrow \text{a = \{11..255\}} & \rightarrow \text{a = \{10..254\}} & \rightarrow \ldots \\
\end{align*}
\]

and while the state space is finite, there are as many multi-states in the reduced system as there were states in the original system. Each multi-state contains a different
set of evaluations of \( a \) and each pair of multi-states is thus different. Furthermore, many states (individual variable evaluations) are represented multiple times: given that states \( s_1 \) and \( s_3 \) in the above system have the same program counter, the values of \( a \) between 10 and 254 are represented twice in these two multi-states alone.

On the other hand, for a specification \( \neg \bigotimes \neg x = 1 \) and a program

\[
x = 1; \text{read } y; \text{while ( true ) } y++; \]

the reduced transition system contains three multi-states and three transitions:

\[
\begin{align*}
x = 0 & \quad y = 0 \\
x = 1 & \quad y = 0 \\
x = 1 & \quad y = \{0..255\}
\end{align*}
\]

whereas the original transition system contains 256 paths that each enclose into a cycle only after 256 unfoldings of the while loop. △

3.2. Explicit-Set Representation

The first representation considered in this paper stores the valid evaluations in an explicit set, enumerating all possible combinations of individual variable evaluations. Hence this approach is closely related to the purely explicit approach of [Barnat et al. 2012] but improves on it in several aspects. Effectively, rather than repeating the verification for every evaluation of input variables (purely explicit approach), with explicit sets we only execute the verification once, from the multi-state containing all initial evaluations. It follows that generating successors needs the following modification: instead of evaluating branching conditions and computing successors only once for the whole state, we must apply these operations to all evaluations comprising the current multi-state. Most importantly, however, deciding state matching is possible on the level of syntax: two states are the same if their representation in memory matches.

Verification with explicit sets clearly retains the biggest limitation of the purely explicit approach, i.e., the spatial complexity grows exponentially with the domains of inputs variables. We postpone discussing the comparison between the explicit set approach and the purely explicit approach until Section 4 and only remark that while the improvement is considerable, it still scales linearly with the domain size. Thus adding a single 32-bit variable increases the time complexity \( 2^{32} \) times. Verifying programs with realistic domains of input variables appears to require a symbolic representation. For this reason, we will focus in the remainder of this section on a representation which uses BV formulae and the satisfiability modulo theories (SMT) procedures for this theory.

3.3. Symbolic Representation

The second representation considered in this paper stores the valid evaluation as a collection of BV formulae. The standard approach is to interpret a formula as the characteristic function of a set. A formula \( \varphi \in \Phi \) then represents a set \( \{ \pi \in \text{sort}(D) | \models_{\text{SAT}} \varphi \land \land x_i = v_i \} \). Generating successors with this representation, however, requires the computation of either pre- or post-conditions, which is non-trivial for many instructions. We will thus use a different approach, popular in symbolic execution [King 1976].

The symbolic representation of the set of evaluations considered in this paper stores the possible current values of variables as images of bit-vector functions. For the following theoretical discussion we set the data part of multi-states \( s(\text{data}) \) to a vector of BV formulae \( (\rho_1, \ldots, \rho_d) \): the transition relation labels collected along the path from \( l_0 \). Such a vector may represent a set of evaluations \( \{ \pi | \models_{\text{SAT}} \rho_d \circ \ldots \circ \rho_1 \land \land x_i = 0 \land \land x'_i = v_i \} \), i.e., the image of the transition relation.
In practice, the transition relations obtained from interpreting program commands are mostly a conjunction of a guard formula \( \gamma \), which refers only to current-state variables, and a set of update formulae \( x' = \varphi \), where \( \varphi \) refers to current-state and input variables. This allows representing the data part of multi-states as a sequence of formulae referring only to input variables. The conjunction of guards forms the so-called path condition and the sequence of updates delimits the values of variables. Thus assuming that a variable \( x \) is associated with a formula \( \varphi \) in a state \( s \) (with path condition \( \delta \)), taking the transition \( s(pc) \xrightarrow{\gamma \land x = \psi} s'(pc) \) creates a state \( s' \) with path condition \( \delta \land \gamma [x \mapsto \varphi] \) and \( y \) is associated with \( \psi [x \mapsto \varphi] \). This representation will not be used in the theoretical part of this paper, but Section 4.1.3 gives details on the experimental implementation which uses the representation.

**Definition 3.9.** The set-symbolic CFG \( G = (S, s_0, T, A) \), similar to the set-reduced CFG limits the transition labels to \( \top \) but unlike the set-reduced CFG, data evaluates to a vector of BV formulae from \( \Phi \). The generation from a CFG \( P = (L, l_0, T, F) \) is defined as follows:

- \( S = \{ s : \{pc, data\} \to L \cup [\Phi] \mid s(pc) \in L, s(data) = (\rho_1, \ldots, \rho_d), \rho_i \in \Phi \} \)
- \( s_0 = \{ (\{pc, l_0\}, (data, (\land x'_i = 0)) \} \}
- \( T = \{ s \to s' \mid s(pc) \xrightarrow{\gamma \land x = \psi} s'(pc), s(data) = (\rho_1, \ldots, \rho_d), \overset{\text{SAT}}{\models} F \rho \circ \rho_d \circ \cdots \circ \rho_1, s'(data) = (\rho_1, \ldots, \rho_d, \rho) \} \)
- \( A = \{ s \in S \mid s(pc) \in F \} \)

Hence a multi-state in the set-symbolic state space consists of a vector of formulae. As stated earlier we want to interpret this vector as the image of the transition relation that results from appending the individual relations in the vector \( s(data) \). For such an interpretation we then prove that paths in the set-reduced CFGs have their equivalent in the set-symbolic CFGs.

**Definition 3.10.** Let \( G = (S, s_0, T, A) \) be a set-symbolic CFG. The image-based interpretation of set-symbolic states is a mapping from \( S \) to \( \{ (v_1, \ldots, v_n) \mid v_i \in \text{sort}(x_i) \} \). For \( s(data) = (\rho_1, \ldots, \rho_d) \) we define it as:

\[
[s] = \{ \tau \mid \overset{\text{SAT}}{\models} F \rho_d \circ \cdots \circ \rho_1 \land x'_i = v_i \}
\]

**Theorem 3.4.** Let \( P = (L, l_0, T, F) \). Under the image-based interpretation, the set-reduced CFG \( G^r = (S^r, s_0^r, T^r, A^r) \) is trace-equivalent to the set-symbolic CFG \( G^s = (S^s, s_0^s, T^s, A^s) \). That is

1. For each path \( s_0^r \to s_1^r \to \ldots \) in \( G^r \) there is a corresponding path \( s_0^s \to s_1^s \to \ldots \) in \( G^s \), such that \( \forall j \in \mathbb{N}, s_j^r(pc) = s_j^s(pc), s_j^r(data) = [s_j^s] \)
2. For each path \( s_0^r \to s_1^r \to \ldots \) in \( G^r \) there is a corresponding path \( s_0^s \to s_1^s \to \ldots \) in \( G^s \), such that \( \forall j \in \mathbb{N}, s_j^r(pc) = s_j^s(pc), s_j^r(data) = [s_j^s] \)

**Proof.** For zero-length paths, the proof is the same for both (1) and (2). In the set-reduced CFG we have \( s_0^r(data) = \{ 0 \} \). In the set-symbolic CFG we have \( [s_0^s] = \{ \tau \mid \overset{\text{SAT}}{\models} F \land x'_i = 0 \land x'_i = v_i \} = \{ 0 \} \).

1. Let \( s_0^s \to \ldots \to s_n^s \to s_{n+1}^s \) be a path in \( G^s \). From the induction hypothesis we have a path \( s_0^r \to \ldots \to s_n^r \) in \( G^r \) such that \( s_n^r(pc) = [s_n^s] \). Let \( s_n^r(pc) \xrightarrow{\gamma} s_{n+1}^r(pc) \) and \( \tau \in s_n^r(data) \) such that \( \overset{\text{SAT}}{\models} F \land x_i = v_i \). Since \( \tau \in [s_n^s] \) we also have \( \overset{\text{SAT}}{\models} F \rho_d \circ \cdots \circ \rho_1 \land x'_i = v_i \) for \( s_n^s(data) = (\rho_1, \ldots, \rho_d) \). Furthermore,
\[\rho_d \circ \cdots \circ \rho_1 \land x'_i = v_i \text{ is equivalent to } \varphi := (\rho_d \circ \cdots \circ \rho_1)[X' \mapsto X^{n+1}] \land x^{n+1} = v_i \]

and

\[\rho \land x = v_i \text{ is equivalent to } \psi := \rho[X \mapsto X^{n+1}] \land x^{n+1} = v_i.\]

The only common variables in \(\varphi\) and \(\psi\) are \(X^{n+1}\) which are bound to the same values and thus \(\models_{\text{SAT}} \rho \circ \rho_d \circ \cdots \circ \rho_1.\)

All that remains is to show that \([s'_{n+1}] = s_{n+1}(\text{data}).\)

\[\forall \tau' \in [s'_{n+1}] \iff \models_{\text{SAT}} \rho \circ \rho_d \circ \cdots \circ \rho_1 \land x'_i = v'_i \]

\[\iff \exists (\tau \in [s'_n]), \models_{\text{SAT}} \rho \circ \rho_d \circ \cdots \circ \rho_1 \land x = v' \land x^{n+1} = v_i \]

\[\iff \exists (\tau \in [s'_n]), \models_{\text{SAT}} \rho \land x' = v'_i \land x_i = v_i \]

\[\iff \tau' \in s'_{n+1}(\text{data})\]

(2) Similarly as in (1), the path in \(G^s\) gives us \(s'_n(\text{pc}) \xrightarrow{\rho'} s'_{n+1}(\text{pc})\) and \(\models_{\text{SAT}} \rho \circ \rho_d \circ \cdots \circ \rho_1.\) Fixing the evaluation of \(X^{n+1}\) variables from the satisfying model to \(\tau\) and renaming the variables we get \(\exists (\tau \in [s'_n]), \models_{\text{SAT}} \rho \land x = v.\) Applying the induction hypothesis we get \(\exists (\tau \in s'_n(\text{data})), \models_{\text{SAT}} \rho \land x_i = v_i.\) Finally, showing that \(s'_{n+1}(\text{data}) = [s'_{n+1}]\) is exactly the same as in (1). \(\square\)

The image-based interpretation crucially depends on the fact that states of set-symbolic CFGs represent sets of program states. These sets of data evaluations are stored compactly in form of \(BV\) formulae but model checking algorithms require distinguishing multi-states. We demonstrate how to compare the explicit part of multi-states efficiently in Section 3.5. Comparing the symbolic part, i.e., performing state matching, requires a different approach for efficient implementation. Below we define a quantified \(BV\) formula which is satisfiable if and only if two given multi-states are different in their data parts. The satisfying assignment of input values for one of the multi-states is such that the resulting evaluation of data variables does not appear in the other state.

**Definition 3.11.** For a set-symbolic CFG \(G = (S, s_0, T, A)\) we define an image-based state matching formula \(\text{image},\) for differentiating two states \(s, s' \in S\) as follows. Let \(s(\text{data}) = (\rho_1, \ldots, \rho_d)\) and \(s'(\text{data}) = (\rho'_1, \ldots, \rho'_d),\) where each variable is also primed, i.e., formulae from \(s'(\text{data})\) refer to \(x'_1, \ldots, x'_m\) and \(\varepsilon_1, \ldots, \varepsilon_n.\)

\[s \nsubseteq s' = \rho_d \circ \cdots \circ \rho_1 \land \bigvee (\varepsilon_1, \ldots, \varepsilon_n), \rho'_d \circ \cdots \circ \rho'_1 \implies \exists x_i \neq x'_i\]

\[\text{image}(s, s') = s \nsubseteq s' \lor s \nsubseteq s\]

**Theorem 3.5.** Let \(s\) and \(s'\) be two set-symbolic states. Then \(\text{image}(s, s')\) is satisfiable if and only if \(s\) and \(s'\) represent two different sets of variable evaluations, i.e., \([s] \neq [s']\).

**Proof.** Let \(\bar{y}'\) be the vector of input values that satisfy \(\text{image}(s, s').\) Given the form of \(\text{image}\) we can assume that \(s \nsubseteq s'.\) Let \(\tau'\) be the satisfying vector of values for variables \(x'_1, \ldots, x'_m.\) From the first conjunct we know that \(\tau' \in [s].\) Let \(\tau \in [s']\) be chosen arbitrarily as well as a vector of input values \(\bar{y}\), one for each \(\varepsilon_i.\) Then \(\rho'_d \circ \cdots \circ \rho'_1 \land \bigwedge \varepsilon'_i = y_j \land x'_i = v'_i\) is unsatisfiable and thus \(\tau \not\in [s].\) If the above formula is satisfiable then the consequent of \(s \nsubseteq s'\) states that in at least one variable evaluation \(\tau'\) differs from \(\tau.\)

Let \(\tau \in [s]\) be such that \(\tau \not\in [s']\) and \(\bar{y}\) be an arbitrary evaluation of \(\varepsilon'_i\) variables. Then either \(\rho'_d \circ \cdots \circ \rho'_1 \land \bigwedge \varepsilon'_i = y_j\) is unsatisfiable or \(\exists x_i \neq x'_i\) holds since \(\tau\) differs from every element of \([s']\). \(\square\)

The above theorem and the existence of SMT solver for the quantified bit-vector theory (QBV) give a feasible way of checking state equivalence under image-based sym-
bolic representation. By extension this allows the implementation model checking for arbitrary system that uses input variables and even those that read from input variables iteratively. The most noteworthy limitation of such an approach is the complexity of the satisfiability checking of $QBV$ formulae, which, in theory, is an $\text{NEXPTIME}$-complete problem. Reported practical results are much more promising but the achieved times are still orders of magnitude higher than equivalent operations in classical model checking.

There are two possible ways of overcoming this obstacle. One possible way is to reformulate the state matching query to allow faster satisfiability checking. The other possible way is to reformulate the definition of state matching that would be correct while easier to check. It can be easily observed that semantic equivalence between $s(data)$ relations is not a correct state differentiating mechanism with respect to set-based reduction. However, given the increase in efficiency of $BV$ (quantifier-free) satisfiability compared to $QBV$ satisfiability, we now describe under what conditions would semantic equivalence be a permissible substitute for the exact, image-comparing, equivalence.

Let us limit our scope to programs that read only a fixed number of times from the input set of values, i.e., $I$ is a finite set of cardinality $m$ and the program statements refer only to input variables among $t_1, \ldots, t_m$. We claim that under such a restriction, it is admissible to define the state matching based on semantic equivalence as follows.

**DEFINITION 3.12.** For a set-symbolic $CFG$ $G = (S, s_0, T, A)$ we define an equivalence-based state matching formula $\text{equiv}$ for differentiating two states $s, s' \in S$ as follows.

$$
\begin{align*}
\text{equiv}(s, s') &= s \not\subseteq s' \lor \bigvee_{i=1}^d \rho_i \not\subseteq x_i \\
\text{equiv}(s, s') &= s \not\subseteq s' \lor s \not\subseteq s' \lor \bigvee_{i=1}^d \rho_i \not\subseteq x_i
\end{align*}
$$

Hence the formula $\text{equiv}$ is a quantifier-free $BV$ formulae and checking its satisfiability is an $\text{NP}$-complete problem. Note that now the satisfying assignment can either come from the last disjunct – is an input variable evaluation that satisfies, say, the left path condition $\rho_h \circ \ldots \circ \rho_1$ but does not satisfy the right path condition – or from the modified subset inclusion. In the latter case, the satisfying assignment satisfies the path condition of $s$ and at the same time leads to different evaluation of program variables by $s'$. The exact meaning of admissibility with respect to state matching is formalised in the following theorem.

**THEOREM 3.6.** Assuming the verified program reads only finitely many times from input, the model checking procedure which uses $\text{equiv}$ for state matching satisfies the following three conditions.

1. **sound:** every counterexample exists also in the set-reduced $CFG$.
2. **complete:** every violating run will eventually be detected.
3. **terminating:** verification of both correct and incorrect programs will terminate.

**PROOF.** (1) Given that the image-based equivalence is strictly weaker than the semantic equivalence, i.e., $\neg \text{equiv} \Rightarrow \neg \text{image}$, every state equivalence located by the semantic equivalence is also located by the image-based equivalence. Consequently, every accepting cycle found using $\text{equiv}$ exists in the set-reduced transition system.

(2) Let us assume there is an accepting cycle in the system, i.e., $s_0 \rightarrow^* s_1 \rightarrow^* s_2$ such that $\text{image}(s_1, s_2)$ is unsatisfiable. Given the restriction above, we know that on the path from $s_1$ to $s_2$ the transition labelling relations does not use input variables. Let us denote these relations as $\rho_1, \ldots, \rho_d$. Their composition $\rho_d \circ \ldots \circ \rho_1$ constitutes a permutation of its domain (otherwise $\text{image}(s_1, s_2)$ would be satisfiable), but not necessarily an identity. In the case the composition is not an identity, $\text{equiv}(s_1, s_2)$ is satisfiable and the computation continues. However, $\rho_d \circ \ldots \circ \rho_1 \circ \rho_d \circ \ldots \circ \rho_1$ is also a permutation of the
same domain and furthermore the evaluation of control variables remains unmodified, and thus the resulting state is accepting if and only if \( s_1 \) was accepting. Finally, given the finite number of possible permutations of a finite domain, we conclude that there exists \( k_0 \leq k_1 \) such that \( (\rho_d \circ \ldots \circ \rho_1)^{k_1-k_0} \) is an identity on the domain of \( (\rho_d \circ \ldots \circ \rho_1)^{k_0} \) and thus \( \text{equiv}(s_{k_1-k_0}, s_{k_1}) \) is satisfiable, where \( s_k \) is the state resulting from \( k+1 \) times iterating the original cycle from \( s_1 \) to \( s_2 \).

(3) The termination can be established simply by observing that there is only finitely many mappings from \( I \) (given the above restriction) to \( \text{sort}(D) \). Then the state space of model checking is also finite even though possibly larger than the original, set-reduced transition system. □

Apart from the restriction to programs that do not read inputs on a cycle, the proposed method utilising semantic equivalence of states may produce larger state spaces than the original state matching based on image equivalence. Note that in the proof of Theorem 3.6 we relied on the number of permutations to be finite, in order to detect existing state equivalences. In practice it means that the traversal of the state space has to visit and store (at most) \( k_1 m \) new states before detecting the equivalence. There \( k_1 \) depends on the particular function composition collected from the cycle transitions and \( m \) is the length of the cycle in the set-reduced system. As the Example 3.3 demonstrated, the value of \( m \) also depends on the transition labelling function. Together, the model checking using semantic equivalence has to unroll the transition relation to reach two fixed points: (1) to stabilise the image set and (2) to stabilise the permutation of that set. We investigate the increased complexity in the number of states and the improvement in the verification time in Section 4.

Example 3.7. To see why the restriction to programs without iterative input reading is necessary, consider this simple program:

\[
a = 0; \quad \text{while}(\text{true}) \quad a = a + \text{read}(\);
\]

The cycle introduces a new input variable with every iteration but more importantly each variable can modify the evaluation of the program variable \( a \). Let us denote by \( s_i \) the states after \( i \) iterations of the cycle, \( s_i(\text{data}) = (\rho_1, \ldots, \rho_{\ell_i}) \), and let \( \ell_j \) be the variable introduced in \( j \)-th iteration. Then the evaluation \( \ell_k = 0, k \in \{1, \ldots, j-1\}, \ell_j = 1 \) always differentiates the last state \( s_j \) from all the preceding states: for all \( k \leq j \) we have that only \( a \to 0 \) satisfies \( \rho_d \circ \ldots \circ \rho_1 \land \ell_k = 0 \) but \( s_j \) evaluates \( a \) to 1.

\[\triangle\]

3.4. Storing and Searching

During its state space traversal, an explicit-state model checker enumerates and stores the visited states in order to detect accepting cycles and to be able to decide that the system is correct (when the whole state space was traversed without detecting any accepting cycles). The question of how to implement an efficient representation of multi-states is at the centre of this paper. (In the following we refer to the objects in a state space as \textit{states}, unless the distinction between multi-states and \textit{single}-states is necessary.) In classical explicit model checking, individual states represent one evaluation of variables and are thus stored as a list of values, each at a fixed position in a block of memory. Postponing any other state reduction techniques for later discussion, states represented explicitly can be efficiently manipulated by copying and comparing contiguous blocks of memory and more importantly compressed using hashing. As a hashing function we consider any function (not necessarily injective) that maps a block of memory to a small number (small compared to the domain of the hashing function, often 32- or 64-bits integer number).

Storing visited states in a hash table allows one to decide whether a state was visited before with an average-time constant complexity. Hashing, however, is only permissi-
ble on a set where every element has a canonical representation and this representation can be computed efficiently. More precisely, it is required that for two elements \( e_1, e_2 \), it holds that if the hashing function \( h \) returns different hashes, \( h(e_1) \neq h(e_2) \), then \( e_1 \) is not the same object as \( e_2 \). This is readily available for sets whose elements are stored explicitly, but for symbolic representation the situation is more complicated.

The set of \( \mathcal{BV} \) functions does not have the property of efficiently accessible canonical representations, i.e., \( \mathcal{BV} \) formulae have canonical forms but the process of canonisation is an NP-complete problem. Clearly, a \( \mathcal{BV} \) function, when stored as its abstract syntax tree is not a canonical representation of its image nor of its semantics. For example, \( 3 + 2 \) and \( 2 + 3 \) both map to 5, yet they are syntactically different and thus their representations do not match. Canonical representations of \( \mathcal{BV} \) formulae exist, but their size grows exponentially in the presence of multiplication, and it is questionable what would be the improvement compared to the explicit set representation. Furthermore, the canonical form of \( \mathcal{BV} \) formulae is canonical only with respect to satisfiability, i.e., two functions have the same canonical representation if and only if they are equisatisfiable. It follows that such canonical form need not preserve the images of functions contained within the original formula.

3.4.1. Linear Storage. Given the basic principle of the combination of explicit and symbolic approaches to model checking, that the control part of individual states is stored explicitly, one still can partially employ hash-based search. Effectively, only the states that share the same control part must be tested for equivalence of the data part in order to decide state matching. The control part is canonical and the standard hash-based search can be used to distinguish states that differ in the control part. This considerably improves the time efficiency, since the number of states among which one must search using the more expensive procedure is smaller. For further discussion let us denote the set of states of \( G = (S, s_0, T, A) \) that share the same control part with a state \( s \) as \( [s]^c \), i.e., \( [s]^c = \{ s' \in S \mid s'(pc) = s(pc) \} \).

A straightforward way of storing the set \( [s]^c \) is in a linear array. Then deciding whether a new \( s' \) is already present in \( [s]^c \) constitutes a similar search as in hash table collisions: the array storing \( [s]^c \) is traversed in a linear fashion and every element tested for equivalence with \( s' \). If such an \( s'' \) is found for which \( \text{image}(s', s'') \) is unsatisfiable then the traversal is aborted (\( s' \) is already stored); otherwise, if \( [s]^c \) was traversed to the end, \( s' \) is appended to \( [s]^c \) as a new state. Note that finding a state \( s' \) to be a new state requires the traversal of the whole \( [s]^c \), yet for every \( s'' \in [s]^c \) the formula \( \text{image}(s', s'') \) is satisfiable. Since finding a satisfying assignment to a \( \mathcal{QBV} \) formula is considerably faster than proving unsatisfiability, it follows that detecting already visited states is more expensive than finding a state to be genuinely new.

There are several possible ways of improving this basic scheme. One can clearly see that the finer the state space is divided solely by the control part the faster the state matching is. This can be utilised by moving some of the data into the control part. For example, if we can be certain that a particular variable can be evaluated to a single value in a state, then it can be safely moved to the control part to be used in further, hash-based, state space division. Another optimisation is to use the satisfying assignments of previously checked \( \text{image} \) formulae. As stated before, such an assignment is in fact an evaluation of input variables that leads to an evaluation of program variables that exists in one state but not in the other. Since the data part of states connected by the same control part was formed under similar conditions (often only a different iteration of a cycle), it might happen that the evaluation that differentiated two such states could differentiate also the next ones. Furthermore, to test whether such evaluation could be reused is possible without calling the SMT solver, using the
explicit evaluation method incorporated in the model checker. We now explain these optimising heuristics in more detail.

3.4.2. Explicating Variables. If some variables do not depend on inputs or if their values were limited by path conditions, it may happen that only one value is permissible as their evaluation (see [Beyer and Löwe 2013] for explicit-value analysis using a similar idea). This can be detected by forcing the satisfiability solver to generate at least two different evaluations: if it fails then only one exists. That evaluation can be employed to faster differentiate states during the hash-based search, since it canonically represents the data. It requires adding another set of control variables $C_D$ to store the explicit values, i.e., for each $x \in D$ we add $x^C$ into $C_D$. Thus hashing is performed on the set $\{s(pc), s(x_1^C), ..., s(x_n^C)\}$ and we add new pairs $(x^C, c \in \text{sort}(x))$ into $s$ when only a single evaluation $e$ of $x$ satisfies the path condition. More precisely, for $s'(\text{data}) = (\rho_1', \ldots, \rho_n')$ when testing satisfiability of $\rho_1 \circ \ldots \circ \rho_1'$, we extract the satisfying evaluation as $\overline{\pi}$. Then all data variables $x_i$ are tested for satisfiability of $\rho_1 \circ \ldots \circ \rho_1' \wedge x_i \neq v_i$. If the previous formula is unsatisfiable we add $(x^C, v_i)$ into $s$.

Explicating variables does not require any further modification of either the accepting cycle detection or database searching. The desired effect of this heuristic is that the number of states with the same explicit part — evaluation of input variables and also a witness — gets reduced. The database thus stores more explicit states, but the complexity of a hash-based search is negligible compared to the satisfiability-based search which resolves hash-based collisions. There is a negative effect, however, in that detecting variables with a single satisfying value requires further calls of the SAT solver. The experiments of Section 4 show that even when the explicating is successful in only a few instances, the overall effect is positive.

3.4.3. Witness-Based Matching Resolution. The advantage of hash-based searching is that the amortised complexity of a single search query can, under certain conditions, be constant. Less efficient storage methods use linear ordering on the elements and tree-like structure where all possible collisions can be detected in a logarithmic number of steps. The following heuristics attempts to simulate the important property of a tree-like structure where all possible collisions can be detected in a logarithmic number of steps. The following heuristics attempts to simulate the important property of a tree-like structure where all possible collisions can be detected in a logarithmic number of steps. The following heuristics attempts to simulate the important property of a tree-like structure where all possible collisions can be detected in a logarithmic number of steps. The following heuristics attempts to simulate the important property of a tree-like structure where all possible collisions can be detected in a logarithmic number of steps.

Two sets of symbolically represented states sharing the same explicit part cannot be distinguished using hashing. Consider the model $Y$ gained from checking satisfiability of $\text{equiv}(s, s')$. It is an evaluation of input variables and also a witness of the difference between $s$ and $s'$ (note that we use equivalence-based state matching). This witness can be reused and can replace some calls of the SAT solver when processing a new state $s''$. For example, let the model from some previous $\text{equiv}(s, s')$ evaluate the input variables to $\overline{\pi}$, such that $\rho_1 \circ \ldots \circ \rho_1' \wedge \bigwedge t_j = y_j$ evaluates the data variables to $\overline{t}$, $\rho_1 \circ \ldots \circ \rho_1' \wedge \bigwedge t_j = y_j$ evaluates the data variables to $\overline{t}'$, and $\overline{\pi} \neq \overline{\pi}'$. Then if $\rho_1'' \circ \ldots \circ \rho_1'' \wedge \bigwedge t_j = y_j$ evaluates the data variables to $\overline{\pi}$, where $\overline{\pi} \neq \overline{\pi}'$, we have $s'' \neq s$, otherwise $s'' = s$. Notice that obtaining $\overline{\pi}'$ and testing the equality with $\overline{\pi}$ can be done without calling the SAT solver since we can simply substitute the input variables $\overline{\pi}$ and evaluate the resulting formulae.

Let us focus the following discussion on a single set of states with the same explicit part. The above observation can be used more extensively as the number of states increases, since every pair among those states constitutes a witness. We propose to build a hierarchy among the states, attaching two sets of states to every witness, denoted as $\{s_1, \ldots, s_i\} \gg y \{z_1, \ldots, z_j\}$, where $\overline{y}$ is the witness of $s_1 \neq z_1$. The following properties hold for each $1 \leq i \leq l$ and $1 \leq j \leq r$:
The witnesses are stored in a priority queue and the one with the highest priority is used for the next state matching. The priority of every witness equals the sum of the number of states that it connects, that is \( l + r \), but we also continually update the priority with every state matching: if a state is found different from states in a set \( Z \), then we remove the states in \( Z \) from both sets connected by each witness. This ensures that, on average, each state matching removes the highest number of states still to be compared with. Once all witnesses with non-zero priority have been tested, the remaining state matchings must be decided with standard calls to the SAT solver.

Similar, though not equivalent simplification can be obtained from reusing the witnesses of image \((s, s')\), i.e., in the case of image-based state matching. There the model is again an evaluation \( \overline{\varphi} \) for the input variables in \( s \) such that for any input evaluation \( s' \), \( s'(\text{data}) \) leads to an evaluation of the state-forming variables different from \( s(\text{data}) \). While this cannot be used to differentiate a new state completely without any call to the SAT solver, one can exponentially decrease the theoretical complexity of the call by checking satisfiability in \( B\mathcal{V} \) instead of in \( Q\mathcal{B}\mathcal{V} \). (Equivalently, one could generalise the idea by stating that reusing a witness allows for differentiation with satisfiability of formulae with one less quantifier alternations.) If the quantifier-free formula \( \rho_d \land \ldots \land \rho_1 \land \rho'_n \lor \ldots \lor \rho'_1 \land \bigwedge x^n_i = x_i \land \bigwedge y^j_k = y_k \) is satisfiable then \( s'' \neq s' \); if it is unsatisfiable then \( s'' = s \).

3.5. Control Explicit—Data Symbolic Model Checking

At this point we have all the necessary methods described to sufficient detail to be able to construct the model checking algorithm using set-based state space reduction. It will prove useful to consider a model checker to consist of four communicating components: (1) an implicit graph description, i.e., the input program, (2) an accepting cycle detection algorithm which continuously generates the explicit graph, (3) a database of already visited states, and (4) a counterexample extractor. Also, even though the model checking will be explained on the control-flow graphs using image-based state matching, it works exactly the same for equivalence-based state matching.

An implicit graph description for the purpose of model checking implements three functions: initial, successors, and is_accepting, that together exhaustively represent a program \( P = (L, l_0, T, F) \).

- The function initial simply returns the initial state of the program, i.e., \( s \) such that \( s(pc) = l_0 \), \( s(\text{data}) = (\top) \).
- The function successors takes a state \( s \) as its parameter and returns a set of all its successors in the reduced transition system (under image-based reduction) generated as described in Definition 3.9, i.e., \( s' \in \text{successors}(s) \iff s \rightarrow s' \). Notice that generating each successor requires calling the SAT solver to verify that \( \rho_d \land \ldots \land \rho_1 \land \rho'_n \lor \ldots \lor \rho'_1 \land \bigwedge x^n_i = x_i \land \bigwedge y^j_k = y_k \) is satisfiable, where \( s(\text{data}) = (\rho_1, \ldots, \rho_d) \) and \( s(pc) \models s'(pc) \in T \). Using the explicit-set representation, computing \( \text{successors}(s) \) requires computing the successors for each \( \pi \in \ll s \rr \).
- Finally, \( \text{is_accepting}(s) \) is \( \top \) if and only if \( s(pc) \in F \) is an accepting state.
Algorithm 1: Generic accepting cycle detection

Input: Program description Graph; Database Visited
Output: A pair $\langle\{\text{Correct}, \text{Incorrect}\}, S\rangle$, where $S$ is a subset of program states that contains a reachable accepting cycle if the first element is Incorrect

1. $\text{Ready}.\text{push}(\langle\text{States}.\text{push}\text{.back}(\text{Graph}.\text{initial}()), \text{nullptr}\rangle)$
2. while $\text{Answer} = \text{Continue}$ do
3.  $\langle s, \pi \rangle := \text{Ready}.\text{pop}()$
4.  if $\neg \text{Visited}.\text{has}(s)$ then
5.    $\text{Succs} := \text{Graph}.\text{successors}(\text{States}[s])$
6.    $\text{SuccRefs} := \text{map}(\text{Succs}, \text{States}.\text{push}\text{.back})$
7.    $\text{Visited}[s] := \langle\text{SuccRefs}, \pi\rangle$
8.    $\langle\text{SuccRefs}, \_\rangle := \text{Visited}[s]$
9.    $\langle\text{Answer}, \text{Next}\rangle := \text{acc}\_\text{cycle}\_\text{step}(s, \text{SuccRefs})$
10.   if $\text{Answer} = \text{Continue}$ then
11.    foreach $s' \in \text{Next}$ do
12.      $\text{Ready}.\text{push}(\langle s', s \rangle)$
13. return $\langle\text{Answer}, \text{Next}\rangle$

The database Visited is an associative array, storing a pair $\langle\text{Succs}, \pi\rangle$ with each state reference. References are denoted by underlining and are implemented as pointers into a linear array States. Hence Succs is a sequence of references to the successors of $s$, and $\pi$ is the reference to the parent state of $s$. In order to support a graph traversal algorithm, the database implements two functions: $\_\text{has}$ and $\_\text{has}$. The semantics of $\_s$ is such that either the pair $\langle\text{Succ}, \pi\rangle$ associated with $s$ is returned or a reference to a new pair is returned if $s$ is not in Visited. The other functions have their expected meaning which is easily inferable from their use in Algorithm 1.

Furthermore, all database functions can be implemented by the generic searching function described in Algorithm 2. Note that the function in Algorithm 2 is essentially a hash-based search with linear conflict resolution. The reason for our describing it to any extent is to expose the difference between the hash-based search for the set of control equal, data conflicting database elements, and the search within this set. While the former is a very fast, syntax-based comparison of two blocks of memory, the latter, represented on line 3 by the call to a SAT solver on one of the image functions, can be extremely slow, since it has to interpret the function representing variable evaluations. As the experiments will later show, the efficiency of the whole model checking approach is crucially dependent on the size of these conflicting sets of states.

The accepting cycle detection algorithm itself, depicted in Algorithm 1, is again a generic graph traversal, abstracted from implementation details of cycle detection algorithms. Details of particular algorithms are not necessary for understanding this work and can be found in the original publication of the respective algorithms. The interested reader should consult [Brim et al. 2004], [Černá and Pelánek 2003], [Courcoubetis et al. 1992] for details on MAP, OWCTY, and Nested DFS algorithms, respectively (those are the three algorithms implemented in DIVINE). The existence of a database of known states is used for three main reasons: (1) it is more expensive to generate successors of a state through implicit graph representation than to obtain them directly from the database, (2) in order to decide correctness of a system the algorithm must traverse the whole graph and thus be able to decide when all states
have been visited, (3) the accepting cycle detection requires that the state matching is implemented. Hence based on (1) we should access the successors of a state using the database (line 8) instead of reconstructing them again (line 5) when the present state is already stored in the database. The output of the accepting cycle detection, in the case the program is incorrect, is a subset of the state space which contains an accepting cycle reachable from the initial state.

One detail of the accepting cycle detection is interesting with respect to the correctness of the overall model checking procedure. When a state is revisited, the algorithm only preserves the older version of the state: the one stored in the database. In standard model checking these two states are identical, but under set-based reduction they may differ in both the path condition and the functions representing variable evaluations. The following theorem shows that in the set-based reduced systems, i.e., with state matching at least as strongly differentiating as the image-based, any version of the state can be used with the same effect. The implications of storing only the oldest version of a state are observable when we try to interpret a path in the system, which happens solely when counterexamples are extracted.

**THEOREM 3.8.** Let $s$ be a reachable state in a set-symbolic $G = (S, s_0, T, A)$ and let $π$ be a path from $s$ to $s'$ such that $\not\models_{\text{SAT}} \text{image}(s, s')$. Then for every path $\xi_1$ from $s'$ to $s_1$ there is a path from $s$ to some $s_2$ such that $\not\models_{\text{SAT}} \text{image}(s_1, s_2)$.

**LEMMA 3.9.** Let $s, s' \in S$ be such that $\not\models_{\text{SAT}} \text{image}(s, s')$ and let $ρ ∈ Φ$ be such that for $r, r' ∈ S$ we have: $r(\text{pc}) = s(\text{pc})$ and $r(\text{data}) = s(\text{data}), ρ; r'(\text{pc}) = s'(\text{pc})$ and $r'(\text{data}) = s'(\text{data}), ρ$. Then $[r] = [r']$.

**PROOF.** Let $s(\text{data}) = (ρ_1, \ldots, ρ_d)$ and $s'(\text{data}) = (ρ'_1, \ldots, ρ'_n)$. Then

$$\begin{align*}
\forall ∈ [r] &\iff \not\models_{\text{SAT}} ρ ⋀ ρ_d ⋀ \ldots ⋀ ρ_1 ⋀ υ_i = v_i \\
&\iff \not\models_{\text{SAT}} ρ ⋀ υ_i ⋀ υ_i = v_i ⋀ \ldots ⋀ ρ_1 ⋀ υ_i = v_i \\
&\iff \not\models_{\text{SAT}} ρ ⋀ υ_i ⋀ υ_i = v_i ⋀ \ldots ⋀ ρ_1 ⋀ υ_i = v_i & \text{from Theorem 3.5} \\
&\iff \not\models_{\text{SAT}} ρ ⋀ ρ_n ⋀ \ldots ⋀ ρ_1 ⋀ υ_i = v_i \\
&\iff \text{∀ } [r] ∈ [r'] \end{align*}$$

□

**PROOF OF THEOREM 3.8.** Let the path from $s_0$ to $s$ be $s_0(\text{pc}) \overset{ρ_1}{\rightarrow} \ldots \overset{ρ_d}{\rightarrow} s(\text{pc})$ and the path from $s$ to $s'$ $s(\text{pc}) \overset{ρ_1}{\rightarrow} \ldots \overset{ρ_n}{\rightarrow} s'(\text{pc})$. We prove the statement by induction on the length $p$ of the path from $s'$ to $s_1$. For $p = 0$ the statement follows from the assumptions. Let $s'(\text{pc}) \overset{\xi_1}{\rightarrow} \ldots \overset{\xi_p}{\rightarrow} s_{p1}(\text{pc}) \overset{\xi_{p+1}}{\rightarrow} s_1(\text{pc})$, then from the induction hypothesis we have a path $s(\text{pc}) \overset{θ_1}{\rightarrow} \ldots \overset{θ_p}{\rightarrow} s_{p2}(\text{pc})$ such that $\not\models_{\text{SAT}} \text{image}(s_{p1}, s_{p2})$. From $\not\models_{\text{SAT}} ζ_{p+1} ⋀ ζ_p ⋀ \ldots ⋀ θ_1 ⋀ ζ_1 ⋀ η_p ⋀ \ldots ⋀ η_1$ we have $\not\models_{\text{SAT}} ζ_{p+1} ⋀ ζ_p ⋀ \ldots ⋀ θ_1 ⋀ η_1$ and thus there is $s_{p2} \rightarrow s_2$. Finally, from Lemma 3.9 it follows that $[s_1] = [s_2]$. □

The subset returned by the Algorithm 1 is not in a form presentable to the user as a counterexample and needs to be further clarified, i.e., the actual counterexample has to be extracted from Next using the Algorithm 3. The algorithm frequently refers to the graph induced on parent edges in order to accelerate the location of both an accepting cycle and the path to that cycle from the initial state. Hence three different graph traversal procedures are used in this algorithm: forward search along normal edges (Reach$_f$), forward search along parent graph edges (Reach$_p$), and backward search along parent graph edges (Reach$_P$). The first three parameters are common to both procedures, i.e., the number of steps (for unlimited traversal), starting vertex,
Algorithm 2: Database searching

**Input**: State $s$; Database Visited; Modifying function update

**Output**: Pair $<\text{Answer}, s>$, where $\text{Answer} \leftrightarrow s \in \text{Visited}$ and $s$ is the reference to the stored state. Furthermore, Visited is correctly modified using function update.

1. $\text{Key} := \text{hash}(s(pc))$
2. **foreach** $s' \in \text{Visited}$ : $\text{hash}(s'(pc)) = \text{Key}$ **do**
3.  
4.  
5. return $<\perp, \text{Visited}.update(s)>$

and the subgraph that limits the scope of the search. The last parameter of the backward search is a set of vertices on which the traversal should be aborted.

**Example** 3.10. We use the relatively simple CFG below to demonstrate the counterexample extraction.

![Diagram showing a simple CFG with states and edges]

The input is the lighter green area, which contains an accepting cycle. In this example, we represent parent graph edges with full lines and the remaining edges with dashed lines. The extraction begins by locating the states with no successors in the parent graph, marked with small rectangle, i.e., the states $s_3, s_4, s_5, s_c$. Next, it identifies the states with an accepting predecessor on the parent graph, marked with red-filled rectangles: $s_4, s_5,$ and $s_c$. For each of those states, here demonstrated on state $s_c$, the algorithm computes the set of successor vertices in the whole graph, marked with a darker blue square. Finally, the accepting cycle is identified as the yellow area on the bottom.

The parent graph is the shortest-path tree and thus contains no cycles. Yet some vertices in the parent graph may have outgoing edges in the full graph that end at one of their parent-graph predecessors. Such vertices cannot have successors in the parent graph. These are first located on line 1 and then filtered on 2 to include only those vertices that have accepting predecessors. Such vertices are then iteratively tested for being on a cycle. Hence the sets $R^*_1$ and $R^*_0$ contain the one-step forward closure and the
full backward closure, respectively, the latter along parent edges. $R_b^*$ formation is also stopped at any vertex from $R_f^1$, which identifies the vertex $s_c$ to be on a cycle. Given that the parent graph is a tree, we know that $R_f^1 \cap R_b^*$ contains at most a single vertex. The cycle $R_b^*$ must contain an accepting vertex $s_a$, from which the final backward traversal towards the initial vertex can be started.

3.5.1. Counterexamples. The ability to produce counterexamples is a crucial property of LTL model checking that must be retained even for our combination with symbolically represented data. In standard, explicit model checking a counterexample is an infinite path that violates the desired property, see Algorithm 3. To the user, it is usually reported as a pair (path to a vertex on an accepting cycle, path from that vertex to itself). With symbolic representation of data we can either select one variable evaluation from among those comprising individual states, or use a part of the symbolic representation to provide the user with more information detailing the violating behaviour than what could be inferred from the explicit counterexample.

We propose to represent the counterexample as a quadruple (path $\pi$ to an accepting state, cycle $\zeta$ on that state, path conditions for $\pi$ and $\zeta$). In the description of paths we are only interested in the choices of the control-flow non-determinism that are required to be taken to steer the computation into and through the violating cycle. The path conditions on the other hand, need only to describe what data values are admissible to get to that cycle. It follows that from such a counterexample the user can get not only one (or a fixed number of) concrete run that violates the specified behaviour but also the precise description of all such runs, represented as the path condition. The negative aspect of symbolic counterexamples is related to the first fixed point search as described in Section 3.1, that can lead to counterexamples exponentially longer than those produced by the standard approach.

Altogether, the counterexample is a quadruple $< \pi, \zeta, s_a(pc), s'_a(pc)>$, where $\pi = s_0, \ldots, s_a$ and $\zeta = s_a, \ldots, s'_a$. Note that distinguishing between $s_a$ and $s'_a$ is only necessary when employing image-based state matching. Given that equivalence-based state matching does not allow reading from input variables on a cycle, the path conditions for $s_a$ and $s'_a$ must be exactly the same. Furthermore, we can use the result of Algorithm 3 without any modifications and simply project the paths to only control-flow information and extract the path condition from $s_a$. The image-based state matching must recompute the path condition for $s'_a$ which is not stored in the database. Yet $s_a$ is stored and we know that $s_a$ is the direct predecessor of $s'_a$ and thus the states stored
Algorithm 4: Database searching with witness-based matching resolution

**Input**: State $s$; Database Visited; Modifying function update

**Output**: Pair $\langle Answer, s \rangle$, where $Answer \leftrightarrow s \in \text{Visited}$ and $s$ is the reference to the stored state. Furthermore, Visited is correctly modified using function update.

1. $Key := \text{hash}(s(pc))$
2. $Visited' := \text{witnessBasedMatchingResolution}(\text{Visited})$
3. $\text{foreach } s' \in \text{Visited}' : \text{hash}(s'(pc)) = Key$ do
4.     if $\not\models_{\text{SAT}} image(s, s')$ then
5.         return $\langle \top, \&s' \rangle$
6.     else
7.         $\omega' := \text{extractWitness}(s, s')$
8.         $\text{updateWitnesses}(\omega')$
9.     return $\langle \bot, \text{Visited.update}(s) \rangle$

in $R_1^1$ have correct path conditions. It follows that $s_a$ in the algorithm already is the correct $s'_a$ for the counterexample.

### 3.5.2. Deadlocks

The above description is complete and correct for arbitrary system that does not contain deadlocks, i.e., at least one guard is satisfiable at any state of the system. In order to extend our approach to systems with deadlocks (which is desirable since deadlock-freedom is an important property to be verified), the algorithm has to correctly resolve the situation when there is a value for which no guard is satisfiable. Formally, let $\Gamma_c$ be the guards on system transitions leaving a state $s$ and $\Gamma_a$ the guards on specification transitions. Then if $\rho_d \circ \ldots \circ \rho_1 \land \bigvee \Gamma_a \land \bigvee \Gamma_p$ is unsatisfiable the traversal algorithm creates $s'$ as a copy of $s$, sets $s'(data)$ to $s(data)$. $\bot$ and stores it as another successor of $s$. This way the ability to detect deadlocks is restored and with it also the standard semantics of LTL on finite runs – implicit appending of a self-loop – is handled correctly.

### 3.5.3. Witnesses

In the case of equivalence-based state matching, the witnesses may take two forms: (1) for demonstrating path condition difference and (2) for demonstrating data evaluation difference. In both cases, the witness contains two sets of state references: one for positive and one for negative instances. Witnesses of form (1) further contain an evaluation of input variables $\overline{y}$, such that for a positive state $s$ it holds that $\overline{y}$ satisfies the path condition, i.e., $\models_{\text{SAT}} \rho_d \circ \ldots \circ \rho_1 \land \bigwedge t_j = y_j$; $\overline{y}$ does not satisfy the path condition for negative states. Witnesses of form (2) contain two sets of evaluations: $\overline{y}$ of input variables and $\overline{v}$ of state forming variables. The idea for witnesses of form (2) is that positive states map $\overline{y}$ to $\overline{v}$, i.e., for a state $s$ it holds that $\rho_d \circ \ldots \circ \rho_1 \land \bigwedge t_j = y_j$ and $\bigwedge x'_i = v_i$. Negative states then map $\overline{y}$ to a vector other than $\overline{v}$.

Using witnesses can be incorporated into database searching (Algorithm 2) as demonstrated in Algorithm 4 into which three additions were made. First, the matching resolution was added on line 1a which prunes the database Visited, removing states using known witnesses while concurrently modifying the priority queue Witnesses (Algorithm 5). The second addition on line 4b extracts the satisfiability model and builds a new witness based on this model (Algorithm 6). The final addition is on line 4c and there we updates the priority keys of witnesses according to the newly added state $\omega'$ (Algorithm 7).
Algorithm 5: witnessBasedMatchingResolution

**Input**: Database Visited
**Output**: Subset \( \text{Visited}' \) of known multi-states

1. \( \text{Visited}' := \text{Visited} \)
2. while \( \neg \text{Witnesses}.\text{empty}() \) do
3.     \( \omega := \text{Witnesses}.\text{pop}() \)
4.     \( \text{res} := \{\exists \text{SAT } \rho_n \circ \ldots \circ \rho_1 \land t_j = \omega.y_j \land \bigwedge x_i = \omega.v_i \} \ ? \text{positive} : \text{negative} \)
5.     \( \text{Visited}' := \text{Visited}' \setminus \omega.\text{res} \)
6.     \( \text{foreach } \omega' \in \text{Witnesses} \) do
7.         \( \omega'.\text{positive} := \omega'.\text{positive} \setminus \omega.\text{res} \)
8.         \( \omega'.\text{negative} := \omega'.\text{negative} \setminus \omega.\text{res} \)
9.         \( \text{key} := |\omega'.\text{positive} \cup \omega'.\text{negative}| \)
10.        if \( \text{key} < \text{threshold} \) then
11.            Witnesses.remove(\( \omega' \))
12.       Witnesses.decreaseKey(\( \omega', \text{key} \))
13.     return \( \text{Visited}' \)

Algorithm 6: extractWitness

**Input**: Differentiable states \( s, s' \)
**Output**: Witness \( \omega' \)

1. \( M := \text{getModel}(\text{image}(s, s')) \)
2. \( \omega'.y := M|I \)
3. \( \omega'.\mathcal{P} := M|\mathcal{D} \)
4. return \( \omega' \)

Algorithm 7: updateWitnesses

**Input**: Witness \( \omega' \)

1. Witnesses.add(\( \omega', 0 \))
2. foreach \( s' \in \text{Visited} \) do
3.     \( \text{res} := \{\exists \text{SAT } \rho_n \circ \ldots \circ \rho_1 \land t_j = \omega.y_j \land \bigwedge x_i = \omega.v_i \} \ ? \text{positive} : \text{negative} \)
4.     \( \omega'.\text{res} := \omega'.\text{res} \cup \{s'\} \)
5.     \( \text{key} := |\omega'.\text{positive} \cup \omega'.\text{negative}| \)
6.     Witness.decreaseKey(\( \omega', \text{key} + 1 \))

4. RESULTS
This section describes the practical aspects of implementing control explicit—data symbolic model checking. We have implemented set-based reduction using both explicit sets and \( \mathcal{BV} \) formulae, the state matching using both the image-based and the equivalence-based techniques, and the hash-table collision resolution accelerated by both the explicating values and the witness-based matching resolution heuristics. All
the proposed and implemented\textsuperscript{1} techniques were experimentally evaluated on models in the DVE modelling language (either adopted from the BEEM [Pelánek 2007] database or hand-written, complex models) and on randomly generated models (which can be easily modified to highlight specific aspects of individual techniques). Finally, we have delegated the satisfiability queries to the Z3 tool [de Moura and Bjørner 2008], which was at the time of writing the only one implementing QBF satisfiability.

The presented experimental results are intended to demonstrate some of the properties of control explicit—data symbolic model checking. Hence, rather than accumulating a larger number of experiments, we select a representative sample and demonstrate relevant details on that sample. If the reader is interested in more real-world examples, we recommend either [Bauch et al. 2014a] for experiments with Simulink diagram or [Bauch et al. 2014b] for experiments with parallel LLVM programs.

4.1. Case Study: DVE Models

The first set of experiments conducted pertains to the native language of DIVINE [Barnat et al. 2013]. Using an existing language allows for a direct comparison with existing verification techniques, in this case with classical explicit-state LTL model checking. Yet since classical explicit-state model checking only verifies closed systems, i.e., without input variables, this technique also had to be slightly modified. The modification consisted of replacing every transition that used input variables with as many new transitions as was the size of the domain of the original input variable. Formally, a transition \( l \xrightarrow{\rho} l' \) is replaced with \( \{ l \xrightarrow{\rho \land \bigwedge i_j = y_j} l' \mid y \in \text{sort}(I) \} \) and we refer to this approach as purely explicit.

4.1.1. DVE Language for Open Systems. The DVE language was established specifically for the design of protocols for communicating systems. There are three basic modelling structures in DVE: processes, states, and transitions. At any given point of time, every process is in one of its states and a change in the system is caused by following a transition from one state to another. Communication between processes is facilitated by global variables or channels, that connect two transitions of different processes such that these two transitions are followed concurrently. Following a transition is conditioned by guard expressions that the source state must satisfy, and entails effects, an assignment modifying the variable evaluation. The LTL specification is merely another process, whose transitions are always followed concurrently with some transition of the system. A comprehensive treatment of the DVE syntax and semantics can be found in [Šiměček 2006].

The DVE language allows using variables of different types and consequently of different sizes. The proposed modification allows specifying which variables are input variables and what their domains are. Explicit states are represented as a function, which assigns a value to each variable and a state to each process. The multi-states used in our setting are composed of two parts. The first, control part is essentially the whole explicit state as defined above. The second, data part introduces a new variable \textit{data} that represent the evaluations of all input variables. Note that, in order to evaluate expression, we need to have an evaluation that assigns a single value to every variable.

\textbf{Example 4.1.} Below is a DVE model for a server-based work distribution protocol. The process \texttt{Control} computes the amount of work and then chooses a worker in a loop. The worker processes continuously decrease the amount of work allotted to them.

\footnote{\textsuperscript{1}Code available at http://anna.fi.muni.cz/~xbauch/code.html.}

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, Article A, Pub. date: January YYY.
The one interesting extension to the standard DVE language is the declaration of variable \textit{ws}. For technical reasons it is declared as an array of 0 elements, but the semantics is such that \textit{ws} is an input variable with \textit{sort}(\textit{ws}) = \{0,...,1000\}. △

### 4.1.2. Set-Based Reduction with Explicit Sets

As Definition 3.1 prescribes, the function \textit{successors} must evaluate the guard and assignment of every available transition for every evaluation in the set comprising the given multi-state. For a multi-state \textit{s} and an LTS transition \textit{s}(pc) \xrightarrow{\rho} l' the generation of successors amounts to (1) computing the set \textit{Valid} = \{\tau \in \textit{s}(data) \mid |_{\textit{SAT}} \rho \land \bigwedge x_i = v_i\} and (2) computing the set \textit{Applied} = \{\tau' \in \textit{sort}(D) \mid \exists(\tau \in \textit{Valid}), \rho \land \bigwedge x_i = v_i \land \bigwedge x_i' = v_i'\}. Then the successor \textit{s}' of \textit{s} with respect to the above transition is such that \textit{s}'(pc) = l' and \textit{s}'(data) = \textit{Applied}. Hence implementing the function \textit{successors} does not require any modification of the underlying model checking mechanisms. The individual evaluations \tau from \textit{s}(data) are assigned to respective variables \(x_i = v_i\) one after another and for each \tau the results of taking the transition is stored in a new explicit set, forming the \textit{s}'(data). The functions \textit{initial} and \textit{is accepting} are not modified at all, provided there is only one initial evaluation. Finally, note that the complications related to the database of visited states are avoided in the explicit sets approach, since these are canonical representations for the sets of evaluations.

**Purely Explicit Approach versus Explicit Sets.** Similar conditions were chosen for the comparison between the purely explicit approach (original D\textsc{iv}\textsc{ine} denoted as \textit{exp}) and the new hybrid approach (set-based reduction using explicit sets denoted as \textit{sym}): the codes were compiled with optimisation option -O2 using GCC version 4.7.2 and ran on a dedicated Linux workstation with 64 core Intel Xeon 7560 @ 2.27GHz and 446GB RAM.

We have conducted a set of experiments pertaining to Peterson's communication protocol. For the purposes of verification, the protocol is usually modelled in such a way that once a process accesses the critical section, it immediately leaves the critical
Fig. 1: Comparison between the purely explicit model checking (exp-states) and the set-based reduction using explicit sets (exp-sets) with respect to both time and space. The verification was executed using 2 and 32 parallel threads, denoted by w2 and w32, respectively.

section, without performing any work. The introduction of input variables allows the model to achieve closer approximation of practical use by simulating some action in
the critical section, however artificial that action might be. Note that this and other models from the BEEM database are abstractions of real programs and a verification engineer had to choose which aspect of the original program or communication protocol was irrelevant. By adding the support for input variables we allow verification of more realistic models and possible even of unmodified programs.

Hence a global input variable \( l \in \{0 \ldots r\} \) as well as an effect \( l = (l + 1) \mod r \) was added to the model. Note that the effect is not biased towards set-based reduction because it forces inclusion of all subsets \( \{0 \ldots r\}, \{1 \ldots r\}, \ldots \) even in the reduced state space. The two plots in Figure 1 report the results of liveness verification of this modified Peterson's protocol. Verifying this protocol is nontrivial and the best parallel algorithm OWCTY [Černá and Pelánek 2003] requires several iterations before it can answer the verification query. The plots clearly show that exp-states cannot scale with the range of input variables \( r \) (the \( x \)-axis) and even when 32 parallel threads were used, verification of a single variable of range \( 0..140 \) required almost 100 seconds. The exp-sets approach scaled markedly better, easily achieving the range up to 10000 with the same spacial complexity that exp-states needed for two orders of magnitude smaller range.

Experiments with other models from the BEEM database gave similar results. Introducing explicit sets had minimal impact on the performance when no data manipulation was present. More importantly, both time and space limitations were reached for significantly larger domains of variables compared to the purely explicit approach. Depending on the complexity of the model the explicit sets allowed for verification with between one and two orders of magnitude larger data sets. We thus conclude that explicit sets are a useful tool to extend the unit testing capacity of a model checker, but realistic data domains require a symbolic representation.

4.1.3. Set-Based Reduction with Symbolic Sets. Following the space-efficient form of state representation used in DIVINE, the states under set-based reduction are also stored as continuous pieces of memory, composed of the explicit and the symbolic part. The \( s(C) \) part of a state includes the program counter and explicated variables and is stored in the same way as in DIVINE and the \( s(data) \) part is appended to it. The boundary between the two parts is known to the database component of DIVINE and thus the searching algorithm correctly hashes only \( s(C) \) and resolves the resulting collisions of the symbolic \( s(data) \) on a semantic level.

One possible way of representing \( s(data) \) is to follow precisely the generating definitions, i.e., to represent both the path condition and evaluation functions as BV formulae over the input variables from \( I \). This representation, however, may grow exponentially as in the following sequence of assignments:

\[
\begin{align*}
x := 1; & \quad x := x + x; \\
x := x + x; & \quad x := x + x; \quad \ldots
\end{align*}
\]

To prevent the above phenomenon the implementation does not substitute symbolic values into new formulae directly, but remembers each evaluation instance as an independent generation of the value of a given variable. Hence the evaluation of assignments decorates each occurrence of every variable with the correct generation, and, internally, the above example is seen as \( x_0 := 1; \ x_1 := x_0 + x_0; \ x_2 := x_1 + x_1; \ \ldots \) and each assignment is stored as a separate formula.

The substitution is, of course, still required but can be postponed until the final formula is presented to the SMT solver. Then one can either introduce the new variables explicitly or by temporary expressions, if they are not to be interpreted as free variables. The latter option for the above example would translate to

\[
\begin{align*}
&\text{let } ((= \ x0 \ 1)) \\
&\text{let } ((= \ x1 (+ \ x0 \ x0))) \\
&\text{let } ((= \ x2 (+ \ x1 \ x1))\ldots))
\end{align*}
\]
and we need to use it when comparing the path conditions of two states \( s_1 \) and \( s_2 \). There the only free variables in the resulting formula must be the input variables and each state has to have differently named variables. Altogether the formula reads

\[
\begin{align*}
\text{let } & ((= x_j \alpha_j[x_0, \ldots, x_{j-1}, t_1, \ldots, t_k]) \ldots \\
\text{let } & ((= y_j \alpha_j[y_0, \ldots, y_{j-1}, t_1, \ldots, t_k]) \ldots \\
& (\text{distinct } \rho_{n_1} \circ \ldots \circ \rho_1[x_j, t_j] \rho_{n_2}^2 \circ \ldots \circ \rho_1^2[y_j, t_j]) \ldots)
\end{align*}
\]

where \( f[\lambda'] \) denotes that only the variables from \( \lambda' \) appear in \( f \). Since \( t_j \) are the only free variables, the satisfying assignment \( \pi \) is such that \( \models_{\text{SAT}} \rho_{n_1}^1 \circ \ldots \circ \rho_1^1 \land \bigwedge t_j = y_j \) and \( \not\models_{\text{SAT}} \rho_{n_2}^2 \circ \ldots \circ \rho_1^2 \land \bigwedge t_j = y_j \) (or vice versa).

Thus while the satisfiability checking of individual path conditions and the image-based state matching can be presented to the SMT solver as a simple sequence of assertions, specifying equivalence-based state matching requires the use of let constructs. The advantage of the former approach, where formulae refer only to input variables, is that it can be performed using the C++ interface of Z3 and can thus be fully integrated into DIVINE, without the need to communicate with Z3 using SMT2 files or pipes.

Calling the SMT solver is required at one more occasion and that is when array elements are referenced using expressions as indices. Each array element is stored as a separate variable but in order to know which element is referenced, the concrete value of the index has to be resolved. This can be easily achieved by requesting the SMT solver to produce a model of the current set of assertions, and then evaluating the index expression on the model.

The final point is the preservation of the ability to accelerate the computation using parallel processing, and the scalability of verification with DIVINE. Given that DIVINE currently distributes the hash function uniformly across the parallel threads, each thread is assigned a subset of the state database, related to some explicit values of \( s(\text{data}) \). Hence the scalability of the verification of set-based reduced systems is limited, compared to the standard verification, only in that the content of \( s(\text{data}) \) cannot be used to further partition the state space between parallel threads. The experiments in Section 4.1.4 show that this limitation is rarely of any consequence, since the distribution using \( s(\text{data}) \) is mostly sufficient. On the other hand, Z3 is only thread-safe for satisfiability checking of quantifier-free formulae and thus the more general image-based state matching cannot be used at all with parallel state space traversal.

### 4.1.4. Equivalence-Based versus Image-Based State Matching

The bulk of our experiments pertain to the work distribution protocol proposed in Example 4.1. The experiments were conducted on a machine with quad-core Xeon 5130 and 16GB of RAM; DIVINE 2.96 and Z3 4.3.1 were compiled with GCC 4.7.3. We have modelled the protocol in DVE for two worker processes with various parameters open for experimentation, viz. the initial work load for the whole system \( ws \), the maximal increase of work for independent processes \( W \), and the number of inputs (determining the initial individual work load) \( i \).

Table I summarises the results for \( ws = 100 \), \( W = 8 \), \( i = 1 \), where the solver used 16 bits for \( BV \) expressions. There the hybrid representation refers to the use of the explicating variables heuristics, the reported times are in seconds, and the memory consumption is in MBs. (The choice of parameter values was decided by an experiment with a 10 minute timeout, i.e., the reported are the largest values for which all 4 experiments finished within 10 minutes; especially the experiments with hybrid representation managed much higher parameter values.)
Table I: Results for the work distribution protocol.

<table>
<thead>
<tr>
<th>Valuation Representation</th>
<th>Pure</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Matching Technique</td>
<td>Image</td>
<td>Equiv</td>
</tr>
<tr>
<td>Number of states</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Number of transitions</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>Number of deadlocks</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Number of control-flow states</td>
<td>5</td>
<td>54</td>
</tr>
<tr>
<td>Average number of collisions</td>
<td>19.4</td>
<td>1.80</td>
</tr>
<tr>
<td>State space distribution</td>
<td>1:2,23:1,24:2</td>
<td></td>
</tr>
<tr>
<td>Path condition time</td>
<td>~3.2</td>
<td></td>
</tr>
<tr>
<td>Path condition calls</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td>State matching time</td>
<td>78.1</td>
<td>49.6</td>
</tr>
<tr>
<td>State matching calls</td>
<td>2123</td>
<td>2308</td>
</tr>
<tr>
<td>Explicit value time</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>Explicit value calls</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>Deadlock detection time</td>
<td>~0.55</td>
<td></td>
</tr>
<tr>
<td>Deadlock detection calls</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>Verification time</td>
<td>81.92</td>
<td>59.59</td>
</tr>
<tr>
<td>Memory consumption</td>
<td>37.6</td>
<td>27.9</td>
</tr>
</tbody>
</table>

The hybrid heuristics allows a number of possible configurations. For example all variables could initially be stored explicitly and only moved to the symbolic data part after being assigned a value which depends on an input. Another option is whether to allow the variables to switch between explicit and symbolic parts multiple times. In this particular set of experiments all variables are initially symbolic: the test whether a variable is explicable is ran after every modification of \( s(data) \). The test is run for each variable (and thus each variable may become symbolic multiple times), and a variable becomes symbolic again after reading from input.

If a number occupies multiple columns then it represents the measured value for all of those columns. The two explicit value rows report the complexity of detecting that a variable can be explicated. Finally, the state space distribution row summarises the ratio among the numbers of multi-states that share a control-flow part (which were hashed to the same value and then distinguished by calling the SMT solver). For example without explicating variables there were only five different control-flow states, but 97 multi-states altogether. Two control-flow states had unique symbolic data (1:2), one control-flow state was common to 23 multi-states (23:1), and two control-flow states were common to 24 multi-states (24:2).

The efficiency of the hybrid representation depends on the number of variables that get assigned a concrete value (or whose evaluation is restricted to one concrete value) and this particular protocol has only a few such variables. Detection of these variables is thus a time consuming operation, but given the improved state space distribution and the reduction in state matching and in overall times, the use of this heuristic is well justified.

The advantage of equivalence-based (equiv) state matching compared to the image-based (image) is also clear from the reported experiments for this protocol and parameter values. Given the higher complexity of image, the distinction grows with the complexity of the expressions, especially with the use of nonlinear arithmetical operations such as multiplication and division. Given the satisfiability checking algorithm used in the SMT solver for quantified theory, proving unsatisfiability is much more complex than proving satisfiability.
On the other hand, even the quantifier-free formulae used in $\text{equiv}$ can generate very hard SMT queries, as shown in Figure 2a. There the higher spikes for $\text{image}$ mostly relate to proving state equality, i.e., the more complex unsatisfiability of a quantified $\mathcal{BV}$ formula, which is in general two orders of magnitude more complex than state inequality. For the equality-based state matching there is hardly any difference between checking equality and inequality. Until the 140th state matching were the runs of $\text{equiv}$ almost negligible but the last 4 comparisons took 88 seconds each. It is also the
case when equiv has to solve additional state matching instances because of its lower
distinguishing power (compared to image).

Since both the variable explication and the equivalence-based state matching are
heuristics, their relative performance will differ based on the verified program. In this
particular configuration, and generally in most of the configurations we have experi-
enced with, the better performing state matching is the equivalence-based and the
better performing symbolic representation is the one explicating variables. While expi-
lication has a reasonably small overhead on programs where it cannot benefit the
verification, i.e., if each variable must be symbolic all the time, the equivalence-based
state matching may entail considerable overhead.

Figure 2b perhaps better illustrates the comparison between image and equiv and
between pure and hybrid representations. The y-axis for the two lower lines are la-
belled on the right-hand side of the figure; it indicates the number of state matching
calls Algorithm 2 needed to decide if a state is already stored in the database, i.e., the
number of times line 3 was executed. The solid line reports the experiments without
explication and the dashed line experiments with the explicating heuristics. Hence
we can see that the heuristics almost eliminated the increase of the number of com-
parisons for larger state spaces. This improvement can be attributed to hash-based
resolution using the explicit part of multi-states, which precedes the calls of the SMT
solver.

The upper lines, measured on the left, report the cumulative time needed by each call
of Algorithm 2. The comparison between image and equiv is less obvious in the average
case, where both techniques require approximately 0.1 seconds to resolve membership
and this time complexity only slowly grows throughout the state space traversal. How-
ever, there is a noticeable difference in the worst case instances, which are two orders
of magnitude more complex than the average instances: a phenomenon which does not
appear for equiv (except the special case in Figure 2a). The most likely explanation
of this observation lies in the implementation of first-order satisfiability. For quantifier-
free theories there are very successful heuristics for both satisfiability and unsatisfia-
bility. At the time of writing, however, the heuristics for proving quantified formulae
unsatisfiable are much less successful, compared to the satisfiable case.
Table II: Five-number summary of SMT queries for the random edge generation.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>q1</th>
<th>med</th>
<th>q3</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding calls</td>
<td>5841</td>
<td>9086</td>
<td>13960</td>
<td>17915</td>
<td>25978</td>
</tr>
<tr>
<td>Equiv fce SAT calls</td>
<td>791</td>
<td>2100</td>
<td>2713</td>
<td>4467</td>
<td>7700</td>
</tr>
<tr>
<td>Equiv fce UNSAT calls</td>
<td>44</td>
<td>79</td>
<td>104</td>
<td>121</td>
<td>160</td>
</tr>
<tr>
<td>Equiv pc SAT calls</td>
<td>0</td>
<td>563</td>
<td>1356</td>
<td>2859</td>
<td>5425</td>
</tr>
<tr>
<td>Equiv pc UNSAT calls</td>
<td>818</td>
<td>2145</td>
<td>2768</td>
<td>4522</td>
<td>7779</td>
</tr>
<tr>
<td>Image SAT calls</td>
<td>1841</td>
<td>2925</td>
<td>5126</td>
<td>6284</td>
<td>8511</td>
</tr>
<tr>
<td>Image UNSAT calls</td>
<td>15</td>
<td>90</td>
<td>118</td>
<td>136</td>
<td>934</td>
</tr>
<tr>
<td>Image UNSAT timeouts</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>Path condition SAT calls</td>
<td>130</td>
<td>176</td>
<td>239</td>
<td>256</td>
<td>288</td>
</tr>
<tr>
<td>Path condition UNSAT calls</td>
<td>212</td>
<td>243</td>
<td>261</td>
<td>324</td>
<td>370</td>
</tr>
<tr>
<td>Longest path condition (B)</td>
<td>8547</td>
<td>17280</td>
<td>29791</td>
<td>58526</td>
<td>728392</td>
</tr>
<tr>
<td>Database size (B)</td>
<td>60066</td>
<td>109374</td>
<td>144402</td>
<td>183968</td>
<td>293120</td>
</tr>
<tr>
<td>Total memory (kB)</td>
<td>12508</td>
<td>14058</td>
<td>15636</td>
<td>18890</td>
<td>1064820</td>
</tr>
</tbody>
</table>

Finally, the last advantage of equiv resides in its parallellisibility, although mainly because of the limitation of Z3, which might be removed in the future. Figure 3 reports the parallel scalability of equiv for 1..4 CPU cores. Theoretically, the scalability is limited by the distribution of concrete states between parallel threads, which can force some threads to solve a larger number of more difficult state matching instances than other threads (for example the 1st thread for 3 cores or the 2nd thread for 4 cores). However, the 2.5x speedup for 4 cores demonstrates a relatively successful preservation of the original scalability of DIVINE, which was one of the primary motivations for this work.

4.2. Case Study: Random Models

In order to investigate the relation between image-based and equivalence-based state matching in more detail, we have implemented a simple generator of transitions of the form $^\gamma(x' = f(x)) \rightarrow$. Hence the influence of the explicit control is completely avoided since every state has the same control part. The arithmetic operations within $f$ were evenly distributed among bitwise and Peano operations, including division and modulo, although this distribution was parametric. Other parameters of the random models were: the number of state-forming variables, the probability of reading from input, the average depth of syntactic trees of both $\forall \exists \gamma$ predicates and functions, etc.

4.2.1. Image versus Equivalence using Linear Search. For the first set of experiments, 500 edges were generated and their effects stored in a database, i.e., one state $s$ among the states in the database was selected and the newly generated edge $l_0 ^\gamma(x' = f(x)) \rightarrow l_0$ was appended to $s$. If $\rho_{d} \circ \ldots \circ \rho_{1} \land \gamma$ is satisfiable, then a new state $s'$, where $s'(data) = s(data).(^\gamma(x' = f))$, is added to the database. Adding to the database requires deciding the membership of $s'$, which was in this set of experiments implemented as a linear search, using both equivalence-based and image-based state matching for comparison. To obtain representative results, the experiment was repeated 100 times, each time with a different seed of randomness.

Table II summarises the statistical distribution of the complexity of these experiments as a sequence of five-number summaries. Each measured value is given in 5 sample percentiles: the smallest observation, the lower quartile, the median, the upper quartile, and the largest observation. The data in the upper part of the table (the top 10 lines) represent the number of call to the SMT solver required during the experi-
We distinguish between satisfiability queries that were satisfiable (SAT) and those that were unsatisfiable (UNSAT). Each query was preceded by the system encoding – translation into Z3 internal format – and each had a time limit of 10 seconds, though only the QBV queries of the image-based state matching timed out. Following the definition of equiv, the equivalence-based state matching is split into two parts: comparing the path conditions (Equiv pc) and comparing the functions (Equiv fce), where comparing path conditions preceded comparing functions.

The first line gives an accurate impression of the penalty for using non-canonical representation. The encoding procedure translates the internal representation of BV formulae into the Z3 format. Calling an SMT solver is an expensive operation and our experiment needed almost 14000 calls on average to revisit a program location 500 times. Another observation is that separating the equivalence-based state matching into path condition equality and evaluation equality may be to the detriment of the total running time. On average almost 3000 checks showed the path condition equal and in these cases the resolution still had to be decided by the subsequent evaluation equality (Equiv fce). The separation would be useful provided the complexity of checking the path condition was much faster. This does not appear to be the case, as the timing results below demonstrate.

Finally, the lower part reports the memory requirements of each experiment. Terms in the database were stored concisely without substitution (see Section 4.1.3), yet encoding into Z3 format required substitution, hence the size of the longest path condition sometimes exceeded the size of the whole database, which itself remained relatively small.

The box plot in Figure 4 is derived from the same set of data, reporting only the time distribution. The plot is divided into three sets of five-number summaries: the first collects the sum of times for individual activities during the insertion of all 500
edges into the database, while the second and third report the distribution of the average and the maximum times over the 100 repetitions. Focusing only on the median values (bold lines) in the average case (the set of plots in the middle), one would conclude that the equivalence-based state matching is only moderately faster. Indeed in a majority of individual SMT queries, the image-based state matching was at most 4 times slower. Yet as the third set demonstrates, the worst case instances are one or two orders of magnitude worse. Table II already showed that 31 unsatisfiable image queries required more than 10 seconds, where the highest number for equivalence was below 0.5 seconds. Nor can we observe that there is a clear bound after which all image queries could be pronounced unsatisfiable, since even the satisfiable queries have peaks close to the 10 second time limit. The clear advantage of the equivalence-based state matching thus is the stability of the time requirements. The disadvantage, apart from limitation to programs with bounded number of inputs, is that the average case performance is merely 4 times better than the image based state matching.

4.2.2. Experiments with Witness-Based Matching Resolution. The final set of experiments demonstrates the efficiency of the witness-based matching resolution using satisfiability models as witnesses of state difference (see Section 3.4.3). The comparison between linear search and witness-based matching resolution is illustrated in Figure 5. The number of SAT queries (Figure 5a) grows linearly for the linear search with the number of seen edges (the upper and lower progression relate approximately to the new state being and not being in the database, respectively). The use of witnesses decreases the number of SAT queries effectively to a constant value, independent on the number of states stored in the database. Yet as Figure 5b demonstrates, the time improvement is only 7-fold, i.e., linear, whereas the number of queries suggested an improvement from linear to a constant complexity. There are two reasons for this relatively modest improvement. First, as clear from Algorithm 4, the overhead of maintaining the witnesses can be nontrivial and grows with their numbers. Second, even though only a constant number of instances need to be resolved by a SAT solver, the unsatisfiable queries (when the new state already resides in the database) cannot be resolved using witnesses. The unsatisfiable queries are commonly more time consuming and thus the resulting improvement is negatively affected since all unsatisfiable queries have to be resolved by a SAT solver anyway.

5. RELATED WORK
There are three vantage points that appear useful for placing the presented work among the established body of verification methodology. First, based on the problem being solved, this paper addresses bit-precise LTL model checking of parallel programs with inputs, where computer variables are interpreted as bit-vectors and not as mathematical integers. Second, based on the achieved results, we have reduced the state space by handling separately the control- and data-flow aspects of parallel programs. And third, based on the distinguishing aspect of our approach, the described verification method combines explicit and symbolic approaches by enumerating the thread interleavings explicitly and by representing variable evaluations as images of bit-vector functions. The related work is analyses based on these three perspectives to locate the differences and common aspects with the content of this work. We have also devoted the first part of this section to shortly describing the symbolic representations most commonly used in verification.

5.1. Symbolic Representations
The symbolic representations employed in formal verification can be divided between canonical and non-canonical. If the object being represented is a Boolean func-
Fig. 5: Step-wise evolution of the state matching time and the number of SAT queries on a random model using equivalence-based state matching comparing the linear search (full) with the witness-based resolution (pruned). The linear search traverses sequentially the set of multi-states that has an equal control part. The witness-based resolution heuristics accelerates the linear search by reusing inequality proofs.
tion then the canonicity of the representation means that for two Boolean functions \( f, f' : \mathbb{B}^n \to \mathbb{B} \) such that any \( n \)-long vector of Boolean values is mapped to the same value by both \( f \) and \( f' \), then the representations of \( f \) and \( f' \) are exactly the same (syntactically equivalent). The definition of canonicity naturally extends to formulae of FO theories, such as the theory of bit-vectors, which are the objects primarily represented in this work.

5.1.1. Canonical Representations. The most important example of canonical representation of Boolean functions are the Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bryant 1986], though in the future discussion we simply refer to them as BDDs. Apart from their other uses, BDDs were crucial in the development of symbolic model checking, given the efficient representation of state spaces they provide [McMillan 1992]. Yet the pivotal bottleneck of classical BDDs is their size when arithmetic operations are represented. It was shown by Bryant that BDDs grow exponentially for integer multiplication [Bryant 1991], regardless of the variable ordering. The discovery of this limitation lead to subsequent modification and alternative diagrams.

Linear functions may be canonically represented by Binary Moment Diagrams (BMDs), which allows efficient representation of digital systems at the word level [Bryant and Chen 1995], but BMDs cannot represent dividers and some Boolean functions representable by BDDs [Becker et al. 1997]. Algebraic Decision Diagrams (ADDs are essentially BDDs with leaves that take values from a set of constants) allow representation of modular arithmetic and are smaller than BMDs [Ravi et al. 1996]. Boolean Expression Diagrams (BEDs extend BDDs with binary Boolean operator vertices) represent any Boolean circuit in linear space, furthermore quantification and substitution (important in fixed point iterations) are also linear [Andersen and Hulgaard 1997]. Finally, the Taylor Expansion Diagrams (TEDs) allow constant computation of multiplication and polynomial computation of bounded exponentiation, yet the wrap-around behaviour of modular arithmetic cannot be represented (for a comprehensive summary of canonical representations see [Ciesielski et al. 2009]).

5.1.2. Non-canonical Representations. The use of non-canonical representations of formulae of FO theories is justified by their potential conciseness, given that all canonical representation grow exponentially for some arithmetic functions [Bryant 1991]. This paper only mentions two representations, one using automata [Boigelot 1999] and the other using monadic logics [Henriksen et al. 1995]; while these representations are very effective in some areas, they were shown impractical for sets of variable evaluations [Wolper and Boigelot 2000]. Similarly, the use of Presburger arithmetic formulae (for their use in verification, see for example [Boigelot and Wolper 1994]) and their combinations with decision diagrams [Bultan et al. 1998] are not detailed on the ground of full Peano arithmetic being used in the verified programs. Finally, given that satisfiability of Peano arithmetic formulae is undecidable, most researchers in program verification focused on using bit-vector theory to represent computation, and designed efficient decision procedures for satisfiability of \( \mathcal{BV} \) formulae.

Early decision procedures for a subset of \( \mathcal{BV} \), i.e., for the quantifier-free fragment with bit-level logical operation, composition, and extraction were based on formula canonisation (using a set of rewriting rules) thus allowing Shostak combination [Shostak 1982] with other theories [Cyrluk et al. 1997]. The set of allowed operations was extended to include Presburger arithmetic in [Barrett et al. 1998]. The possibility of deciding satisfiability for non-fixed size bit-vectors was investigated in [Büchi and Segerberg 1998; Pichora 2003]. A solution based on solving the linear problems using Müller-Seidl’s algorithm and the non-linear problems using Newton’s \( p \)-adic iteration algorithm was implemented in [Babic and Musuvathi 2005]. This combination...
allows for the Nelson-Oppen combination without converting the entire problem into a SAT instance (also called flattening).

Precise modular arithmetic was first considered in [Wang 2006] which reduced the satisfiability of modular arithmetic formulae over the finite ring $\mathbb{Z}_{2^\omega}$ to integer programming. For linear modular arithmetic only a linear number of constraints is needed; for non-linear one needs an additional factor of $\omega$ (though the constraints are still linear). The lazy approach, i.e., postponing the flattening process and using first the structural information such as equalities and arithmetic functions, was followed in [Bruttomesso et al. 2007]. Deciding satisfiability based on alternatively improving under- and over-approximation of the original formula was investigated in [Bryant et al. 2007]. Finally, [Wintersteiger et al. 2013] implemented a decision procedure for the quantified bit-vector theory by extending the quantifier-free procedure with model-based quantifier instantiation.

5.2. Bit-Precise LTL Model Checking of Parallel Programs with Inputs

All related methods of program verification are instances of static analysis, in that they verify correctness of some aspect of the program (here specified as an LTL formula) without executing the program. We concentrate on three specific instances: model checking, symbolic execution, and abstraction-based program analysis. One outstanding research direction which is related but does not fit the above categories is the study of value-passing transition systems. Defined in [Lin 1996] these transition systems have edges labelled with guarded assignments and allow bisimulation of two systems to be reduced to finding the greatest solution of the underlying predicate equation system.

5.2.1. Model Checking. Explicit-state LTL model checking [Vardi and Wolper 1986] does not have any theoretical limitation with respect to parallel programs with inputs other than state space explosion since the arithmetic operations are computed explicitly. Given its theoretical basis in $\omega$-regular automata, explicit model checking is local in the sense that only those parts of the system are considered that are relevant to the temporal property being verified. It follows the tableau-proof processes of deciding validity of LTL formulae, where the sequent rules describe what hypotheses must hold for the conclusion to hold (and the hypotheses are often structurally simpler). However, even this limited part of the transition system of parallel programs is often outside the storage capacity of present-day (and arguably even future) computers.

Yet most approaches trying to ameliorate state space explosion, such as partial-order or symmetry reduction and parallelisation (see for example [Barnat et al. 2001]), focus on the control-flow part; ignoring the variable evaluation or assuming it to be fixed. Only rarely was the impact of data-flow on explicit model checking studied [Eisner and Peled 2002] and the results were unfavourable. The recent attempts at incorporating a systematic treatment of input variables into explicit model checking, culminating in this paper, started by introducing a new process into the verified system that generated the possible variable evaluations [Barnat et al. 2012].

Symbolic CTL model checking [Clarke et al. 1986], on the other hand, is global in the sense that the part of the system being represented at some step of the verification may be unreachable from the initial state and thus unnecessary to be considered. This seeming redundancy stems from the representation being symbolic: it is more efficient to incorporate some redundancy than to describe what is redundant and that it should be excluded from the representation. Unlike in local model checking, in global model checking one starts from those states that satisfy the simplest (atomic) subformulae of the specification formula and then structurally progresses towards the computation of the set of states that satisfy the whole formula. If this set contains all initial states of
the program’s transition system then the program is correct. Similarly as with explicit-state model checking, verifying parallel programs with inputs using symbolic model checking is only limited by the complexity of the task. The classical approach based on BDDs [McMillan 1992] suffers from the unmanageable size of BDDs caused by (1) arithmetic operations in the data flow and (2) irregularity of the control flow.

Using non-canonical representations, Boolean Expression Diagrams [Williams et al. 2000] or propositional logic formulae [Bjesse 1999; McMillan 2002], to handle the first cause of BDD size increase, is limited by the introduction of quantifiers when pre- and post-images are computed during the fixed point search. (Note that our research has met similar difficulty when discovering accepting cycles.) Bounded symbolic model checking [Biere et al. 1999a] continues with using logical formulae instead of BDDs and adopts the locality of the explicit approach thus improving also the second cause of size complexity for the price of being incomplete. The completeness was then introduced in [McMillan 2003] and in [de Moura et al. 2003], using Craig interpolants and $k$-induction, respectively. Recently, this approach was improved in [Bradley 2011] by replacing the transition relation unfolding by incrementally generating clauses that approximate the reachability information, thus further accelerating the verification process.

Finally, module checking [Kupferman and Vardi 1996; Vardi 2001] allows verification of systems that interacts with an environment that introduces non-deterministic behaviour. The environment can restrict the set of possible choices of some inputs and the verification should be robust with respect to these restrictions, yet this can only be distinguished by branching time logics; where LTL is linear and its validity thus unaffected by these restrictions.

5.2.2. Symbolic execution. The basic idea of symbolic execution [King 1976; Coen-Porisini et al. 2001] is to track the execution of a program by building its execution tree. The nodes of such a tree consist of a path condition (constraint on the input variables that leads the execution to this node) and symbolic variable evaluation. The symbolic variable evaluation does not interpret the expressions assigned to variables during execution, they are syntactically substituted with the symbolic values of variables they refer to. This verification approach allows for efficient safety analysis of programs with inputs, since only one execution is necessary to consider all possible executions. An extension of symbolic execution to LTL verification was investigated in [Braione et al. 2008] but did not incorporated full LTL on the grounds of state matching not being decidable for theories with unbounded integers.

Without state matching, symbolic execution is not guaranteed to terminate given that the program under verification may contain loops. One possible extension of symbolic execution is the addition of state subsumption [Xie et al. 2005; Anand et al. 2006] (effectively subset equality of concrete evaluations) which improves efficiency but not the scope of supported specifications. Furthermore, the original limitation of symbolic execution to verification of sequential programs was lifted in [Khurshid et al. 2003] by employing a model checker to generate possible thread interleavings.

5.2.3. Abstraction-Based Program Analysis. The verification methods based on abstract interpretation [Cousot and Cousot 1977] do not track the execution of the program under verification in the precision specified in the program. Instead the precision is approximated to a model that can be more easily manipulated, each concrete value of variables mapped to an abstract object, e.g., an interval, that allows simpler and faster computation. The application of abstract interpretation in static analysis has lead to a number of results on real-world, industrial scale programs.

Predicate abstraction for temporal properties was described in [Podelski and Rybalchenko 2005] which builds on the idea of encoding temporal properties into fair
termination. Later, abstraction-based verification of parallel programs was proposed in [Gupta et al. 2011]. Finally, the last ingredient for bit-precise verification was an abstract domain forming a lattice over bit-vectors. One possibility, extending the polyhedra abstraction with implicit wrap-around of arithmetic operations, was investigated in [Simon and King 2007].

5.3. State Space Reduction by Control-Flow Aggregation

Cimatti et al. [Cimatti et al. 2001] combine the approaches from explicit and symbolic model checking by representing subsets of the set of states as individual BDDs (uncertainty states) by relying on forward, hash-based space traversal used in explicit model checking. Uncertainty states are modified according to the transitions applicable to the states that comprise it. Another approach, investigated in [Hungar et al. 1995; Bohn et al. 1998], handles the problem of data-flow non-determinism by solving first-order CTL model checking (given that pre- and post-image computation introduces quantifiers, the first-order extension of a temporal logic is sufficiently powerful to represent the problem). From a semantic point of view, their algorithm iteratively approximates the sets of evaluations annotating nodes of a product of two transition systems: one is essentially the control-flow graph, the other connects related subformulae of the input formula. This verification is incomplete for unbounded data domains unless the data and control variables can be clearly separated. A similar, though still incomplete, approach is to accumulate the path condition during verification that distinguishes between program counter predicates and program variable assertions [Gunter and Peled 2005]. Finally, [Biere et al. 1999b] showed how local and global approaches could be combined by aggregating the sets of states satisfying a subformula in the tableau-based verification.

Another possible aggregation is to form sets from those states with similar observable events, together building an observation graph. Originally described in [Haddad et al. 2004] this approach was optimised in [Baarir and Duret-Lutz 2007] by constructing powersets using previously formed subsets. The limitation of the original paper of the next-free fragment of LTL was recently lifted in [Duret-Lutz et al. 2011].

5.4. Combinations of Explicit and Symbolic Approaches

In [Beyer and Löwe 2013] the authors combine explicit-value analysis with various symbolic approaches, such as abstraction, counterexample-guided refinement, and interpolation to build an abstract reachability graph, extending the work from [Beyer and Keremoglu 2011]. The explicit analysis only tracks those variables necessary to refute the infeasibility of error paths and variables are added in the precision set iteratively, in further CEGAR iterations. The abstraction maps variables to \(\mathbb{Z} \cup \{\top, \bot\}\), \(\top\) for unknown value and \(\bot\) for no satisfiable value and this abstract variable assignment represents the region of concrete data states for which it is valid. Hence the abstraction clusters together sets of states with the same explicit value of every variable.

In [Cimatti et al. 2012] a verification method for SystemC designs was implemented by translating them to multi-threaded programs with cooperative scheduling, shared variables, and mutually exclusive thread execution. In this model only one thread is running at a time, when it stops the scheduler selects the next thread to run. The explicit scheduler, symbolic thread combination uses symbolic abstraction for sequential execution of individual threads while the scheduling policy is handled explicitly. A different combination was investigated in [Sebastiani et al. 2005] which symbolically represented the system state space and explicitly the property automaton. Also related is the concolic execution, see for example [Sen et al. 2005], where the standard symbolic execution is, after a number of symbolic steps, supplemented with a single
step of concrete execution (on a smaller subset of values). Then the symbolic execution continues, maintaining a much smaller representation.

Perhaps the most closely related work is that of [Cook et al. 2005] where the distinction between data- and control-flow non-determinisms is realised via a SAT-based model checker for concurrent (with a finite number of threads) Boolean programs. Data-flow non-determinism is handled by a combination of symbolic model checking and fixed point detection and the control-flow non-determinism is handled by symbolic partial-order reduction. The approach is limited to Boolean programs, yet standard programs can be converted to Boolean using predicate abstraction. Our work is precise with respect to the bit-vector semantics of programs since we do now use any abstraction. Finally, [Cook et al. 2005] verify reachability properties and thus do not require exact state matching. Their approach is thus much faster, but does not allow checking LTL properties.

6. CONCLUSION

Moving from the theoretical application of LTL model checking — a methodology to verify correctness of reactive systems — to practical application on parallel programs requires handling efficiently both the control-flow and the data-flow choices the programs can make. And while a majority of approaches treats these two sources of non-determinism equally, this paper promotes the appreciation of the fundamental differences between control and data flows. We argue that the nature of control flow prevents verification polynomial in the number of program threads (see [Farzan et al. 2013] for current development), and thus we propose to handle the control-flow state space explosion by efficient parallelisation. The data flow, however, is often regular enough for symbolic representations to capture it concisely. Hence the object of our investigation: control explicit—data symbolic model checking.

The combination of explicit and symbolic approaches to model checking presents several possible directions of research and this paper follows the path where a symbolic representation of data is devised, which an explicit model checker could use without requiring modifications. The proposed set-based state space reduction is compliant with this requirement, though at the price of potential inefficiency. After being set-reduced, the state space contains states with explicit and symbolic parts, the latter representing a set of variable evaluations. Hence the crux of the proposed combination: what symbolic representation would be most efficient in representing a set of variable evaluations.

Given the advancement of modern SAT and SMT solvers, and given the potential inefficiency of established canonical representations [Bryant 1991], we opted for the representation based on bit-vector formulae: a bit-vector function represents the set that forms its image. Computing the effect of program statements thus translates trivially to syntactic substitution. Deciding state matching and the related search among already visited states, crucial operations for LTL verification, proved considerably more complicated. We have devised two implementations of state matching and two heuristics that accelerate searching and storing within the database of states. All these implementations were compared with each other and with classical, purely explicit model checking on DVE models of communicating protocols and randomly generated models (allowing more targeted experiments). The common observation is that, with the proposed combination, explicit model checking can now verify programs with non-trivial data flow, which was previously not possible.

The primary purpose of this paper is to investigate control explicit—data symbolic model checking theoretically, with the experimental evaluation undertaken on small models of artificial programs. This allowed us to more accurately pinpoint the limits of the method but does not substantiate our claims of applicability to real world...
problems. To that extend we would like to refer the reader to papers that apply the set-based reduction to realistic programs: Simulink diagrams [Bauch et al. 2014a] and C++ programs via the LLVM bitcode language [Bauch et al. 2014b].

Our investigation also exposed a number of potential directions for future research and open problems. It should, for example, be experimentally validated that BDDs and other symbolic representation are less efficient than SMT solvers for model checking of programs over bit-vectors. One prominent limitation of our approach was the need to precisely distinguish the sets of variable evaluations, hence it would be reasonable to pinpoint the extent to which LTL model checking requires precise distinction of evaluations. Finally, modern program analysis techniques such as counterexample-guided refinement, loop acceleration, compositional approach to control flow, etc. could potentially improve the presented verification method.

REFERENCES


P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis

A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. 2001. Using Symbolic Execution for Verifying Safety-

Model Checking

E. Clarke, O. Grumberg, and D. Peled. 1999. Model Checking

A. Cimatti, M. Roveri, and P. Bertoli. 2001. Searching Powerset Automata by Combining Explicit-

I. Cerna and R. Pelanek. 2003. Distributed Explicit Fair Cycle Detection (Set Based Approach). In


ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, Article A, Pub. date: January YYYY.


