Model Checking Parallel Programs with Inputs

Jiří Barnat, Petr Bauch, Vojtěch Havel
Masaryk University

Abstract—Verification of parallel programs with input variables represents a significant and well-motivated challenge. This paper addresses the challenge with a verification method that combines explicit and symbolic approaches to the state space representation. The state matching between non-canonical representations proved to be the bottleneck of such a combination, since its computation entailed deciding satisfiability of quantified bit-vector formulae. This limitation is here addressed by an alternative state matching, based on quantifier-free satisfiability, and a heuristics optimising the state space searching. The experimental evaluation shows that the alternative state matching causes only a minor increase in the number of states and that, in combination with the heuristics, it considerably extends the scope of applicability of the proposed LTL model checking.

I. INTRODUCTION

The use of parallelism often requires verification of properties that cannot be specified as reachability of specific bad states. For communication and cooperation of multiple processes, the correct functioning requires a certain level of fairness [15] in the consumption of computational resources. More generally any property that can only be refuted by a counterexample of infinite length is outside the scope of verification techniques based on reachability [9]. Such properties can in many cases be specified in a temporal logic, such as Linear Temporal Logic (LTL), for which verification is still feasible [16].

Consider a motivating example of a server that distributes work to a fixed number of client processes. The server does not know in advance the size of any individual work load that it will have to send to each client. Furthermore, the function that chooses which client will be sent the next batch is determined by a hash function, which uses arithmetic operation such as multiplication and modulo, to improve the load balancing. Finally, the developers want to verify that a situation never arises in which one client has a work load above \( x \) but another client will never work in the future. That requirement can be easily captured by the LTL formula

\[ G(\text{load}[\text{client}_1] > x \Rightarrow F \text{ client}_2.\text{work}) \]

If the program under verification reads from nondeterministic inputs, e.g. an input file, sensor chip, etc., its state space quickly grows outside the scope of current techniques for LTL verification. The familiar state space explosion caused by the interleaving of parallel programs is multiplied by the possible evaluations of input data variables. In this paper we combine explicit state model checking to handle the control flow of parallel programs with symbolic representations of data. Yet standard data representations used in symbolic model checking, such as Binary Decision Diagrams [13], grow exponentially in size in the presence of bit-vector (modular) arithmetic over integers [7]. Thus alternative representations of data are necessary to cope with parallel programs reading from inputs.

One possible solution is not to interpret the right-hand sides of assignments (bit-vector functions) but to store them as representations of possible variable evaluations which form the images of these functions. Temporal verification has to detect recurrence of the violating behaviour and thus requires some form of state matching, which proved to be particularly complex to decide algorithmically [3]. Another limitation of using bit-vector functions as data representation is their non-canonicity. Given that two syntactically different functions can represent the same set of values prohibits the straightforward use of hash-based searching, which further decelerates the model checking process. This paper addresses the above limitations by the following contributions.

1) Alternative State Matching: A major limitation of [3] was the prohibitively complicated state matching procedure which required deciding satisfiability of quantified bit-vector formulae. This limitation is here addressed by an alternative state matching procedure using quantifier-free satisfiability. Though not functionally equivalent, we have proved that on a restricted set of input programs (those reading only a bounded number of times from inputs) the new state matching produces correct results.

2) Accelerated State Space Searching: Another limitation, the number of state matching tests required to decide if a state has already been visited, is ameliorated by a heuristics that transfers variables with only one possible concrete evaluation into the explicit part of the state. This heuristics and the above mentioned techniques are evaluated on a set of experiments which demonstrate the progress of LTL model checking towards real parallel programs with bit-vector arithmetic.

A. Related Work

There is a number of verification procedures that allow LTL verification of parallel programs with inputs. The standard explicit-state model checking [16] has to deal with the state-space explosion but the complications related to modular arithmetic are avoided by the explicit computation of program expressions. Symbolic model checking [7] can also solve the problem provided that the symbolic representation employed supports modular arithmetic (we will discuss such representations shortly). Similarly, bounded model checking [5] can be unbounded by a number of approaches: interpolation [14], \( k \)-induction [12], incremental induction strengthening [6]; which provides the method with the ability to decide correctness of the system under verification. Yet the work most closely related to ours is that of Cook et al. [10] where Boolean programs are
verified by control explicit—data symbolic reachability analysis. Our work can be seen as an extension of [10] with support for LTL properties and precise reasoning about bit-vector formulae.

II. PRELIMINARIES

Assuming that a program consists of \( n \) parallel processes, its state is uniquely defined by the evaluation of the local variables \( L_1, \ldots, L_n \) of each process and the global variables \( G \). The domain of these variables is the set of fix-width bit-vectors (bv) for vectors of \( q \) bits, i.e. for \( L_i = \{ x_1^i, x_2^i, \ldots \} \) the domain \( \text{sort}(x_i^q) = \{ 0, \ldots, 2^q - 1 \} =: \text{bv}^q \). We also introduce the set of input variables, denoted as \( I = \{ i_1, i_2, \ldots \} \) that do not contribute to the forming of program states, but that are evaluated randomly, with \( \text{sort}(i_j) \subseteq \text{bv}^q \) for some \( q \).

A process is a triple \( P = (S, \rightarrow, s_0) \), where \( S \) is the set of program counters, a transition relation \( \rightarrow \subseteq S \times \text{Cond} \times (G \cup L_i) \times \text{Act} \times S \) and \( s_0 \in S \) is the initial location. The \( \rightarrow \) relation is an abstraction of both program conditional statements and assignments. Let \( BV \) denote the quantifier-free fragment of the first-order bit-vector theory and \( p_X (f_X) \) a \( BV \) predicate (function) over the variables from the set \( X \). For easier exposition we will abbreviate sets of variables in the following way: \( GL_A = \cup_{i=1}^{n} L_i \), and so on. Then \( Cond \) is a formula \( p_{GL_A} \) that must be satisfied for this transition to be taken and \( Act \) a function \( f_{GL, z} \) that is evaluated and its value assigned to a variable. Put together we have \( (s, p_{GL_A}, x, f_{GL, z}, s') \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \right
\[ \text{diff}(\varphi_F, \varphi'_F) := nseq(\varphi_F, \varphi'_F) \lor nseq(\varphi'_F, \varphi_F) \] which is satisfiable iff \( \varphi_F \) and \( \varphi'_F \) are different states, where \( nseq(\varphi_F, \varphi'_F) \) is defined as \( \varphi_F(pc) \land \left( \bigvee_{x_i \in GL} \varphi_F(d)(x_i) \neq \varphi'_F(d)(x_i) \right) \).

\[ \text{nseq}(\varphi_F, \varphi'_F) := \bigvee_{x_i \in GL} \varphi_F(d)(x_i) \neq \varphi'_F(d)(x_i). \]

and let \( \text{diff}'(\varphi_F, \varphi'_F) := \varphi_F(pc) \neq \varphi'_F(pc) \lor \text{nseq}(\varphi_F, \varphi'_F) \lor \text{nseq}'(\varphi_F, \varphi'_F). \]

Hence the formula \( \text{diff}' \) is a quantifier-free \( BV \) formula and checking its satisfiability is an \( N \text{P} \)-complete problem. The first disjunction witnesses that there is an input variable evaluation that satisfies, say, \( \varphi_F(pc) \) but does not satisfy \( \varphi'_F(pc) \). In the other two disjuncts, the satisfying assignment lies in the domain of the \( \varphi_F(d) \) function (satisfies \( \varphi_F(pc) \)) and at the same time is sent to different values of program variables by the \( \varphi'_F(d) \) function. What exactly is meant by calling the new definition admissible is that a model checking procedure using \( nseq' \) for state matching may find different violating runs and may need to traverse large state spaces. For the formalisation and proof of this statement, please see the full version [2] of this paper.

IV. STORING AND SEARCHING

As noted in the introduction, a model checking with states represented as images of \( BV \) functions cannot use hash tables to implement the database of already visited states. Such a database is crucial for both the underlying accepting cycle detection algorithm and for the termination of the verification process. The reason why \( BV \) functions stored as abstract syntactic trees cannot be distinguished by hashing is that this representation of a set of evaluation is not canonical. More precisely it does not hold that given the inequality of hashes of two states \( h(e_1) \neq h(e_2) \) \( e_1 \) must differ from \( e_2 \). And even though canonical representations of \( BV \) formulae exist, their size grows exponentially [7] in the presence of multiplication.

1) Collision Resolution by Chaining: Given the basic principle of the combination between 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 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 improves the time efficiency, since the number of states among which one must search using the more expensive procedure is smaller. On the other hand, the non-canonicity of the representation does not straightforwardly provide any procedure of distinguishing states with the same control part. Hence every element of the hash table is an array of states sharing the same control part, and searching among those requires a linear number of state matching instances.

2) Early State Distinction by Explicating Variables: If some variables do not depend on inputs or their values were limited by path conditions it may happen that only one value is permissible as their evaluation. This can be detected by investigating the model of satisfiability and employed by hash-based state distinction and requires the addition of another set of variables \( X \) to store the explicit values. Thus hashing is performed on the \( \varphi|\_\text{SAX} \) part of a state \( \varphi \) and we add new pairs (variable, value) into \( X \) when only a single value satisfies the path condition. More precisely, when testing satisfiability of the successor’s path condition \( \varphi'_F(pc) \) we extract the model \( \mathcal{M} \). Then both the global and local variables, say \( x \), are tested for satisfiability of \( \varphi'_F(pc) \land x \neq \mathcal{M}.e\text{val}(x) \). If the formula is unsatisfiable we add \( (x, \mathcal{M}.e\text{val}(x)) \) into \( X \).

V. RESULTS

In order to be able to compare the newly proposed equivalence-based state matching and related heuristics on a sufficiently large set of experiments, we have completely reimplemented\(^1\) the previous method [3] and incorporated it inside DiVinE [4]. The bulk of our experiments pertains to the work distribution protocol proposed in the introduction. 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 compile 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 addition of work for individual

This paper further advances the progress towards bit-precise LTL verification of parallel programs with inputs. The previously reported limitations, caused by state matching based on \(\forall \mathbf{BV}\) satisfiability, are partially transcended by reformulation into a simpler satisfiability problem. Even though the newly proposed equivalence-based state matching procedure is restricted to programs that perform only a bounded number of input reading, there still is a number of problems that are not tractable by other verification methods and the verification of which is accelerated using the new state matching. This we have demonstrated on the case study of a work distribution protocol, where bit-precision is crucial for correctness. The new state matching as well as the explicating values heuristics improve the verification times and also preserve the parallel scalability of the underlying model checker.

### References


