Dynamic Guiding of Bounded Property Checking

Prakash M. Peranandam, Roland J. Weiss, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel
University of Tübingen
Department of Computer Engineering
Email: {prakash,weissr,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de
Category: Formal Verification Methods

Abstract— **Current statistics attribute up to 75% of the overall design costs of digital hardware and embedded system development to the verification task. In recent years, the trend to augment functional with formal verification tries to alleviate this problem. Efficient property checking algorithms allow automatic verification of middle-sized designs nowadays. However, the steadily increasing design sizes still leave verification the major bottleneck, because formal methodologies do not yet scale to very large designs.**

In this paper we present the formal verification tool SymC based on forward state space traversal and so-called AR-automata for property checking, both internally represented with BDDs. Furthermore, we introduce a new methodology called *dynamic guiding*. This methodology best suits multi-module concurrent Finite State Machine (FSM) designs. The aim of guiding is to reduce the intermediate and final BDD size, which in turn makes this verification technique applicable to larger designs. Our approach exploits abstract information of the design in the form of regular expressions and effectively guides the symbolic traversal depending on the verified property.

I. Introduction

Gaining confidence on the correctness of digital designs is one of the major tasks in the system design flow. Steadily increasing design sizes make verification a bottleneck in modern design flows of digital hardware and embedded software systems. State of the art verification techniques include traditional simulation on the one end of the spectrum and model checking on the other end. Simulation is applicable to large designs, whereas model checking provides full error coverage of the design with an automatic process. The most noteworthy contribution in this area is bounded model checking (BMC) [1], which is well suited for finding shallow errors in large designs. All these techniques have their own advantages and disadvantages. State space explosion is the major problem in all formal verification tools.

In this paper we deal with invariant checking and its improvement. For invariant checking, all reachable states of the system are calculated and the desired property is checked to hold. In case of failure a counterexample is generated. The computational complexity of property checking is disclosed when large systems are analyzed. Even utilization of BDD based symbolic techniques does not allow the complete analysis of the state space. Though BMC and BDD based bounded property checking reduce this problem considerably by restricting the state space with in a bound, sometimes intermediate steps are too huge to handle.

Our approach can be clearly separated into two parts, first, bounded property checking using symbolic state space traversal and second, *dynamic guiding* for providing information used during symbolic execution of the system model. The target designs of *dynamic guiding* are concurrent, multi-module, control intensive protocol FSM designs. In contrast to a classical model checker, our bounded property checker SymC performs one forward image computation at a time, i.e. the current set of states is replaced by the set of states reachable from the current set within one transition. This results in an efficient verification for properties with large time bounds by avoiding fixpoint iterations and reachable state set computations [2].

With dynamic guiding we attempt to reduce the BDD size during symbolic state space traversal. Current guiding techniques are static by nature, i.e. the whole verification environment is specialized for a given property. In [3], [4] the authors either constrain the input or remove the uninteresting parts of the design. In contrast, our approach first extracts structural information from every module of the system design. These individual information of modules are then used to generate the guiding sequence for the whole product machine for a given property. Thus, the system model is left untouched and can be used for different properties. Only during symbolic execution the guiding information steers traversal into paths that are more likely to satisfy the checked property. This implies that given the structural information, a guiding sequence can be generated for any property and used during symbolic execution, hence the name dynamic guiding. In [5] a similar concept is used, but in a totally different verification approach in comparison to our work.

The main features of our approach are:

- We present a combination of bounded property checking with dynamic guiding. This combination results in an efficient verification of properties by avoiding fixpoint iterations and reachable state set computations. We examine states reachable up to a given time bound that is given explicitly or implicitly by the property.
- Symbolic forward traversal of the design allows steering the simulation in certain directions, thus avoiding uninteresting parts of the state space. Structural information captured with regular expressions enable creation of guiding sequences tailored for the checked properties.

This work is sponsored by the German Research Grant (DFG projects KOMFORT and GRASP).

The rest of the paper is structured as follows: In the next section we describe our bounded property checker. Thereafter, we detail the dynamic guiding approach. Finally, we conclude and point to ongoing and future work.

II. Bounded Property Checking

In [2] we proposed a formal verification technique for time bounded property checking. The technique performs forward image computation for state traversal, a characteristic shared by forward model checking [6]. Experimental results [2] show that this approach outperforms other property checking methods for certain classes of systems and properties.

The input to SymC is a system description given either as Verilog gatelist or in a simple SMV-like finite state description language. Properties are specified with FLTL [7] or PSL [8] formulas, therefore a tight integration with other property checking tools is provided. The temporal logic formulas are converted to special FSMs, so-called AR-automata [7], which can then be used in the symbolic execution phase. The system description is translated into a finite state system encoded with BDDs. During the symbolic execution we observe the state of our properties and we report success or failure to the user. Figure 1 shows the structure of SymC. The dashed components constitute the new guiding unit and will be described in Section III.

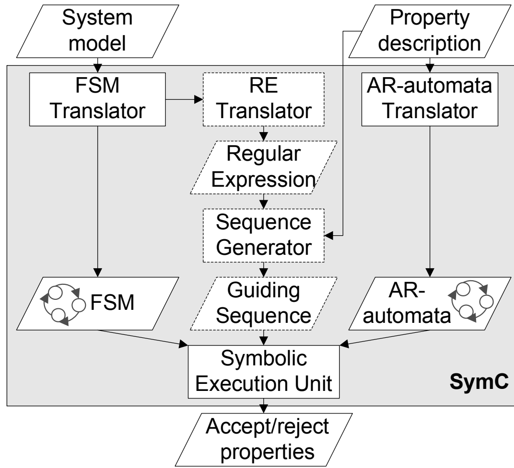

Fig. 1. Overview of SymC, including the optional guiding unit.

The following algorithm sketches the core of the execution engine for one AR-automaton. The algorithm can be extended to multiple AR-automata.

```
// Build product state of the system and the AR-Automaton.
current = start ∧ AR.start;
for ( t = 0 ... k )
  // Compute image of AR-Automaton.
  current = image_AR(current);
  if (check universally)
    if ( current ∧ AR.reject ≠ false ) reportFailure();
    if ( current ∧ AR.accept = current ) reportAcceptance();
  if (check existentially)
    if ( current ∧ AR.accept ≠ false ) reportAcceptance();
    if ( current ∧ AR.reject = current ) reportFailure();
  // Guide the symbolic state traversal into a certain direction.
  current = current ∧ goIntoDirection();
  current = image_T(current);
```

The tool includes a number of optimization techniques. One of them partitions the current state set upon reaching a threshold size. Then, the partition sets are explored sequentially. If a property's AR-automaton reaches either the accept or reject state, exploration of the remaining sets can be skipped, saving time and space. The first version of SymC partitions the state set randomly. At this point, we introduce dynamic guiding in order to select partitions that will make termination of the symbolic execution more likely based on information extracted from both the design and the property. Notice that the guiding unit does not interfere with other optimizations present in SymC.

III. Dynamic Guiding

We identify guiding of a symbolic state space traversal with the information provided for steering the direction of traversal into the interesting state space. Interesting state in our context can be defined as a state which satisfies a property that is expected to hold at least in one path from initial state (existential). If the property has to hold in all paths from initial state (universal), then we look for states that falsify the property. Steering the traversal into interesting state implies that we aim at reducing the intermediate memory and the time to verify a property. The goal of our approach is to get an guiding mechanism for different properties for the same initial condition of the design, and without changing the verification setup. These requirements force the guiding mechanism to have information of the design. The following elaborations rely on the deterministic FSM (D-FSM) formalism.

Definition 1: A D-FSM $\mathcal{A}$ is a 5-tuple $\mathcal{A} = (S, \Sigma, T, s, F)$, where $S = \{s_1, \ldots, s_n\}$ is a finite set of states, Σ is a finite alphabet, $T : S \times \Sigma \to S$ is a transition function, $s \in S$ is the initial state, $F \subseteq S$ is a set of final states.

A short example should illustrate the notion of guiding a FSM. We write $s_i \xrightarrow{a} s_j$ to express that there is a transition from s_i to s_j labeled with a. Consider a FSM which includes the transitions $s_1 \xrightarrow{a} s_2$ and $s_1 \xrightarrow{b} s_3$, where $s_1, s_2, s_3 \in S$ and $a, b \in \Sigma$. Assume that our property involves only symbol a, which discloses that the transition with symbol b is not interesting for our verification. So the traversal can be guided into interesting states by asserting symbol a, which reduces the possible next states of s_1 to s_2 only. Asserting input symbols in a D-FSM means that we are restricting the next state possibilities to one defined state. This restriction of image computation is the key point of guiding in our approach. In order to realize this guiding mechanism, we have to decide on the exact time step for guiding and the specific assertions at that time point. For effective guiding of arbitrary properties, the guiding mechanism should generate assertions to be applied at all time steps during image computation.

Typically, SymC traverses the FSM of the system model from the initial state at time point zero and continues until the

maximum time bound of the property is reached. This unveils that the guiding mechanism needs a sequence of input symbols from time point zero until the maximum bound in the property in order to guide the tool to the interesting state.

Definition 2: Let $\mathcal{A} = (S, \Sigma, T, s, F)$ be a D-FSM, w a string over Σ. $\mathcal{A}$ accepts language $\mathcal{L}(\mathcal{A}) = \{w \mid \delta(s, w) \in F\}$, where $\delta(q, w) : S \times w \rightarrow S$ is an extended transition function that returns the state that is reached when $\mathcal{A}$ starts from q and processes w.

The sequence that is to be generated by the guiding mechanism has to be always one of the words w that is accepted by the $\mathcal{L}(\mathcal{A})$. Such a sequence can be generated using abstract information from the design. In our context abstract information means the regular structure of the language $\mathcal{L}(\mathcal{A})$. This regular structure is represented in the form of regular expressions (REs) as it is the standard and compact way of representing languages. A RE is a string that describes a language, according to certain syntax rules.

Definition 3: A RE consists of constants and operators that denote sets of strings and operations over these sets, respectively. Given a finite alphabet Σ the following constants are defined:

- (empty set) ϕ denoting the set ϕ,
- (empty string) ϵ denoting the set $\{\epsilon\}$ and
- (literal character) $\mathsf{a} \in \Sigma$ denoting the set $\{\mathsf{a}\}$;

as well as the following operations:

- (concatenation) RS denoting the set $\{\alpha\beta \mid \alpha \in R$ and $\beta \in S\}$,
- (set union) $R \cup S$ denoting the set union of R and S,
- (Kleene star) R^* denoting the smallest superset of R that contains ϵ and is closed under string concatenation.

IV. Extraction of Regular Expressions and Guiding Sequences

We will now describe how REs are generated from the input FSMs, and how these are used to generate guiding sequences for the automatic guiding process.

A. Extraction of Regular Expressions

The state elimination method is a standard algorithm in automata theory to convert a D-FSM into a RE [9]. The actual state elimination method generates a RE that is equivalent to a given FSM. This RE represents all words accepted by $\mathcal{L}(\mathcal{A})$ (see definition 2). But we are interested only in the regular structure of the accepted language. So we use a variant of the state elimination method. Figure 2 depicts the basic construction of REs.

RE construction starts from the initial state and collects the symbols along the transistions to next states until it reaches the termination condition as shown in figure 2. The termination conditions of RE construction are that either one reaches an initial state, or a state that is not an initial state and has no further transistions to other new states.

Because we want to deal with system descriptions that can provide multiple initial states s_i, we apply the state elimination method for all initial states to the given FSM and union the resulting REs in one RE $\mathcal{R}$:

$\mathcal{R} = \forall i : \bigcup \mathcal{R}_i \mid s_i \in SI$, where $SI \subseteq S$ is a set of inital states.

In other words, if we have more than one initial states, the procedure has to loop for every single initial state and join them by union. $\mathcal{R}$ is then used to extract the sequence for the property. The REs extracted from the small example in Figure 3 with s_1 as initial state is shown below:

$(\neg req)^* \cup req\ (wait)^*\ \neg wait\ ack$

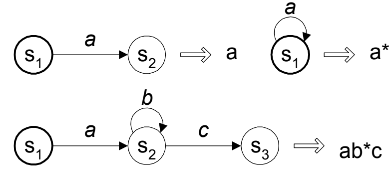

Fig. 2. Examples of FSM to RE translations, where s_1, s_2, s_3 are states, and a, b, c are symbols.

B. Extraction of Guiding Sequences

This section focuses on the extraction of a guiding sequence from the system's RE $\mathcal{R}$ tailored for the given properties. Presently we handle only FLTL formulas of the form A and $A \rightarrow B$, where A and B are FLTL formulas.

Definition 4: Let $\mathcal{A} = (S, \Sigma, T, s, F)$ be a D-FSM. Then syntax of FLTL is recursively defined over the input alphabet Σ:
$\phi := a \mid \neg\phi \mid \phi \wedge \psi \mid X_{[m]}\, \phi \mid F_{[m,n]}\, \phi \mid G_{[m,n]}\, \phi$, with $a \in \Sigma$, $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$.

The FLTL property that is to be verified can be viewed as a discrete representation of the guiding sequence in a special syntax. In principle, verification of a FLTL formula is finding a sequence (word) accepted by the design's FSM that can satisfy the discrete conditions. Such a word is the guiding sequence in our case. For obvious reasons we do bottom-up searching to form the guiding sequence out of REs for the property. The REs resemble all the possible regular structures of the language accepted by the FSM. This is the key point in our sequence generation approach. With a bottom-up search the symbols of the commitment formula B are first located in $\mathcal{R}$. Then, all possible predecessor symbols from that particular RE are collected at each time point.

This process should be carried out from the maximum time bound until time point zero. The ordered collection of symbols from the maximum time bound until time point zero are the possible guiding sequences. The sequence that is generated can be restricted to the condition of the assumption and the temporal operators of FLTL.

The following algorithm sketches the core of the sequence generation engine. Here k is the maximal time bound in the property, c is the commitment signal, a is the assumption signal, and bound is the time bound of the property.

```
// Collect list of commitment symbols.
list-of-symbols.insert(c);
```

```
for ( t = 0 ... k )
    while ( ¬list-of-symbols.empty() )
        // Get all prev. symbols before c from RE.
        c = list-of-symbols.back();
        list-of-symbols.pop-back();
        new-list-of-symbols = get-previous-symbols(c);
        // Update it in the map to produce the sequence.
        M[c] = new-list-of-symbols;
        // Append it to temp list for next loop.
        T = append(T, new-list-of-symbols);
    list-of-symbols = T;
// gen-sequence builds all ordered collections of symbols.
// Map M stores all involved symbols.
gen-sequence(M, a, bound);
```

To illustrate the above algorithm we use a small arbitration example (see figure 3), where the FLTL property is $req \rightarrow F[2]\ ack$:

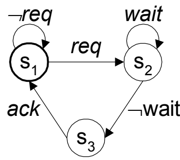

Fig. 3. Simple FSM example.

In a bottom-up search we first locate *ack*, then we search for all possible predecessor symbols to *ack*. In this example only ¬*wait* can occur. Then, we locate the predecessor symbols to ¬*wait*, which are either *req* or *wait*, and so on. This information is stored in a map as follows:

$ack \longmapsto \neg wait$, $\neg wait \longmapsto wait \mid req$, $wait \longmapsto wait \mid req$, $req \longmapsto \neg req \mid ack$ and $\neg req \longmapsto \neg req \mid ack$.

Finally, function gen-sequence generates all possible valid sequences as follows. As the commitment signal is *ack*, it inserts symbol *ack* at the front of the list, then the possible previous symbol ¬*wait* is added in front of the list. Then the next possible successor symbols *wait* or *req* are inserted in front of the list. Trivially, there are two possible sequences:

$Sequence_1 = \{\ req, \neg wait, ack\}$
$Sequence_2 = \{\ wait, \neg wait, ack\}$.

Now the timing condition is checked, where the property instructs that *req* should be at time point 0 and *ack* should be at time point 2. Considering that the sequence elements start from time point 0, the valid sequence as per the condition is $Sequence_1$, which in turn will be used for dynamic guiding. For example if SymC requires guiding at time point 2, the dynamic guiding will assert signal ¬*wait* in order to guide to the success state.

V. Experimental Results

To examine the benefits and the limitations of our ideas, we conducted experiments with our prototype tool with single and multi-module systems. The result was promising for multi module designs, in which every module is of relatively small size and blows up when producting it. Apparently, it was

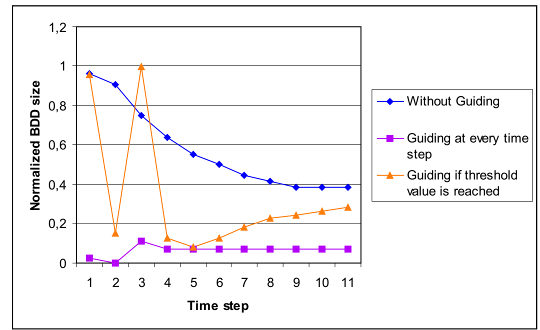

Fig. 4. Comparison of Intermediate BDD Size.

not so convincing for large single module design. To show the benefits of our idea, presently we considered a wireless protocol design, which is restricted to six modules modeling only the connection establishment of the priority flows. Figure 4 shows the BDD size difference for three environments: a) guiding at every time step, b) guiding only if the BDD size is larger than a threshold value, and c) without guiding. The BDD size in the graph is normalized to 1. The graph highlights the fact that our approach reduces the intermediate BDD size, which in turn helps the verification tool to handle designs more efficiently.

VI. Conclusions

We presented a technique for guiding forward state space traversal in a BDD based bounded property checker. It can dynamically guide the traversal for any property with similar initial conditions. This is due to the fact that we extract abstract guiding information out of the real design in a discrete manner. Our tool is successfully tested with some examples. Future work focuses on extending our prototype tool to handle a wider variety of standard designs.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” in *Highly Dependable Software*, ser. Advances in Computers, M. Zelkowitz, Ed. Academic Press, 2003, vol. 58.

[2] J. Ruf, P. M. Peranandam, T. Kropf, and W. Rosenstiel, “Bounded property checking with symbolic simulation,” in *Forum on Specifi cation and Design Languages*, 2003.

[3] A. Casavant, A. Gupta, S. Liu, A. Mukaiyama, K. Wakabayashi, and P. Ashar, “Property-specific witness graph generation for guided simulation,” in *Design Automation and Test in Europe (DATE)*, March 2001.

[4] L. Pierre and T. Kropf, Eds., *Hints to Accelerate Symbolic Traversal*, ser. Lecture Notes in Computer Science, vol. 1703. Springer, 1999.

[5] K. Fisler and P. Kurshan, “Verifying VHDL designs with COSPAN,” in *Formal Hardware Verifi cation*, T. Kropf, Ed., 1997, pp. 206–247.

[6] H. Iwashita and T. Nakata, “Forward model checking techniques oriented to buggy designs,” in *Proceedings of the 1997 IEEE/ACM International Conference on CAD*. ACM and IEEE Computer Society Press, 1997, pp. 400–4004.

[7] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel, “Simulation-guided property checking based on a multi-valued AR-automata,” in *Design, Automation and Test in Europe, DATE 2001*, W. Nebel and A. Jerraya, Eds. IEEE Press, 2001, pp. 742–748.

[8] *Property Specifi cation Language Reference Manual*. Accellera, April 25 2003, version 1.01.

[9] J. D. Ullman, “Stanford lecture notes,” http://www-db.stanford.edu/ ullman/ialc/jdu-slides.html, 2000.