Abstraction-Based Livelock/Deadlock Checking for Hardware Verification

In-Ho Moon and Kevin Harer
Synopsys Inc.
{mooni, kevinh}@synopsys.com

ABSTRACT

Livelock/deadlock is a well known and important problem in both hardware and software systems. In hardware verification, a livelock is a situation where the state of a design changes within only a smaller subset of the states reachable from the initial states of the design. Deadlock is a special case in which there is only one state in a livelock. However, livelock/deadlock checking has never been actively used in hardware verification in practice, mainly due to the complexity of the computation which involves finding strongly connected components.

This paper presents a practical abstraction-based livelock/deadlock checking algorithm for hardware verification. The proposed livelock/deadlock checking works on FSMs rather than the whole design. For each FSM, we make an abstract machine of manageable size from the cone of influence of the FSM. Once a livelock is found on an abstract machine, the livelock is justified on the concrete machine with trace concretization. Experimental results shows that the proposed abstraction-based livelock checking finds real livelock errors in industrial designs.

1. INTRODUCTION

Livelock/deadlock is a well known and important problem in both hardware and software systems. In hardware verification, a livelock is a situation where the state of a design changes within only a subset of the states reachable from the initial states of the design. In a state transition graph, a livelock is a set of states from which there is no path going to any other states that are reachable from the initial state. Since deadlock is a special case in which there is only one state in a livelock, deadlock checking can be done by livelock checking. Thus, livelock implies both livelock and deadlock in this paper. However, livelock checking[1] has never been actively used in hardware verification in practice, mainly due to the complexity of the computation which involves finding SCCs (Strongly Connected Components). Thus, livelock checking has been on hardware designer's wish list to verify their designs.

There have been many approaches on finding SCCs [13, 27, 28, 3, 20, 12]. Among these work, Xie and Beeral proposed a symbolic method finding *terminal SCCs* (in short, TSCCs) using BDDs (Binary Decision Diagrams [4]) in [27]. In a state transition graph, a TSCC is an SCC that does not have any outgoing edges to any state outside the SCC. Thus, a TSCC becomes a livelock group when the TSCC has any incoming edges to the SCC in the state transition graph representing a hardware design. However, even though the method in [27] is an improved method from its previous work [13] in symbolic approaches, it is still infeasible to apply the method to the industrial designs, simply due to the capacity problem of symbolic methods. In general, any BDD-based method can handle only up to several hundred latches without any abstraction or approximation techniques, whereas there can be millions of latches in industrial designs.

In this paper, we first present an improved BDD-based algorithm finding TSCCs from[27] in the following aspects. First, initial state is taken into account in finding TSCCs. Especially, the improved algorithm handles multiple initial states efficiently. Secondly, unreachable TSCCs are distinguished from reachable TSCCs which are more interesting to designers. Thirdly, we provide more intuitive state classification as main, transient, and livelock groups as opposed to transient and recurrence classes in [27]. In our classification, a set of transient states is further classified into main and transient groups. Recurrence class in [27] is mapped into livelock group in our classification. Main group is an SCC that contains the initial state. Transient group is a set of states that belong to neither main nor livelock group. There is one or zero main group in a design per one initial state.

This paper also presents a practical approach for checking livelock using abstraction techniques. The proposed livelock checking works on FSMs(Finite State Machines)[2] rather than the whole design. For each FSM, we make an abstract machine (by localization reduction [15]) of manageable size by the improved BDD method from the COI(Cone of Influence) of the FSM. Once a livelock is found on an abstract machine, the livelock is justified on the concrete machine with trace concretization using SAT (Satisfiability[9]) and simulation. When there is no livelock on the abstract machine, there is no guarantee for no livelock on the concrete machine. However, the bigger the abstract size is, the more confidence we have that no livelock exists on the concrete machine. The key benefit of this abstraction-based livelock checking is that it enables finding real livelock groups that cannot be found by tackling whole design directly.

Once an FSM is given, its COI is first computed. Then, an abstract machine is computed by finding N *influential latches* from the COI. Influential latches are the latches that are likely related with the FSM. N is either pre-defined or a user-defined number of latches in the abstract machine, or

[1]Livelock checking is different from liveness checking and the difference will be explained in Section 2.3.

[2]FSMs are either automatically extracted [26] or any sets of sequential elements that are user-specified.

gradually increased. In general, N is up to a few hundred latches. Influential latches are computed mainly by approximate state decomposition [6]. However, in many cases, the size of COIs is too big for even approximate state decomposition. Thus, a structural abstraction is applied by using connectivity and sequential depth before approximate state decomposition. This structural abstraction reduces the COI to a manageable size by approximate state decomposition.

There is another important hardware property called *toggle deadlock*. A state variable has a toggle deadlock if the state variable initially toggles and the state variable becomes a constant after a certain number of transitions. However, notice that this is not a constant variable since it initially toggles. Toggle deadlock may or may not happen depending on input stimuli in simulation. Therefore, toggle deadlock is also an important property to check with formal approaches.

Experimental results shows that the proposed abstraction-based approach finds real livelock and toggle deadlock errors from industrial designs.

The contributions of this paper are in the three aspects.

- Improved algorithm for livelock checking
  The proposed algorithm improved the existing algorithm [27] in many aspects, such as providing new state classification with initial state, handling multiple initial states, refining the search space efficiently with care states, early termination, and trimming out transient states.
- Abstraction-based livelock checking
  This paper presents theories and an implementation on abstraction-based livelock checking to handle large designs in practice.
- Toggle deadlock checking
  To the best of our knowledge, this paper presents the first method to solve toggle deadlock problem.

The remainder of this paper is organized as follows. Section 2 briefly recapitulates finding SCCs and TSCCs, and describes related work. Section 3 describes our improved algorithm for finding TSCCs. Section 4 explains how livelock is checked on FSM using abstraction. Section 5 describes how to check toggle deadlocks. Experimental results are presented and discussed in Section 6. We conclude with Section 7.

2. PRELIMINARIES

2.1 Finding SCCs

Given a graph, $G = (V, E)$ where G is an infinite transition system of the Kripke structure [8], V is a finite set of states and $E \subseteq V \times V$ is the set of edges, a strongly connected component (SCC) is a maximal set of state $U \subseteq V$ such that for every pair $(u, v) \in U$, u and v are reachable from each other, that is, u is reachable from v and v is reachable from u [27, 28].

Finding SCCs has a variety of applications in formal verification such as Buchi emptiness [11], LTL model checking [25], CTL model checking with fairness constraints [10], Liveness checking [16, 1], and so on.

The traditional approach to find SCCs is to use Tarjan's method [23]. Since this method manipulates the states of the graph explicitly, even though it runs in linear time in the size of the graph, the size of the graph grows exponentially as the number of state variables grows.

To overcome this state explosion problem in explicit algorithms, there have been many publications on symbolic algorithms. Ravi *et al.* [20] provided a taxonomy of those symbolic algorithms. One is SCC-hull algorithms (without enumerating SCCs) [11, 14, 24], and the other is SCC enumeration algorithms [28, 3, 12, 20]. The details are in [20].

2.2 Finding TSCCs

Even though TSCCs are a subset of SCCs in the states of a design, the algorithms for finding TSCCs can be significantly optimized since not all SCCs are of interest.

This section recapitulates the work on finding TSCCs by Xie and Beeral [27]. This algorithm classifies all states into either recurrence or transient class. Recurrence class is a set of TSCCs and the rest belongs to transient class. Let S be the set of states. With $i, j \in S$, $i \to j$ denotes that there is at least one path from i to j. Definition 1 defines *forward set* and *backward set* of a state.

DEFINITION 1. *The forward set of state $i \in S$, denoted by $F(i)$, is the set of states that have a path from i. That is, $F(i) = \{j \in S \mid i \to j\}$. Similarly, the backward set of state i, denoted by $B(i)$, is the set of states that have a path to i. That is, $B(i) = \{j \in S \mid j \to i\}$.*

LEMMA 1. *Let $i, j \in S$. If $j \in F(i)$, then $F(j) \subseteq F(i)$. Similarly, if $j \in B(i)$, then $B(j) \subseteq B(i)$.*

THEOREM 1. *A state $i \in S$ is recurrent if and only if $F(i) \subseteq B(i)$. In other words, i is transient if and only if $F(i) \nsubseteq B(i)$.*

THEOREM 2. *If state $i \in S$ is transient, then states in $B(i)$ are all transient. If state i is recurrent, on the other hand, states in $F(i)$ are all recurrent. In the latter case, set $F(i)$ is a recurrence class, and set $B(i)\backslash F(i)$ (if not empty) contains only transient states.*

Lemma 1, Theorem 1 and 2 are from [27]. Lemma 1 shows a subset relation between two forward sets as well as two backward sets when j is in either $F(i)$ or $B(i)$. Theorem 1 and 2 show how a state is determined whether the state belongs to either recurrence or transient class. Based on Theorem 1 and 2, all TSCCs can be found by performing forward and backward reachability iteratively. The detailed algorithm can be found in [27] and our improved algorithm is described in Section 3.2 with the comparisons to the original algorithm.

2.3 Related work

There are two types of properties in model checking; safety and liveness properties [16]. A safety property represents 'something bad never happens', whereas a liveness property represents 'something good eventually happens'. Liveness checking with a liveness property can be performed by finding SCCs [20]. Liveness checking can also be performed by safety checking with proper transformations [1].

Livelock checking is different from liveness checking in the sense that liveness checking requires a liveness property to work on a design, whereas livelock checking does not require any property and works on a design directly.

There have been many publications on finding all SCCs [24, 28, 3, 12]. Even though all TSCCs can be found by any of these approaches on finding all SCCs, it is not necessary to find all SCCs for finding all TSCCs since we are interested in finding only all TSCCs for livelock checking.

Hachtel *et al.* [13] proposed a symbolic approach to find all recurrence classes concurrently identifying all TSCCs by computing *transitive closure* [17] on the transition graph with the Markov chain. Due to the complexity of transitive closure, this approach takes significantly more time and memory than a reachability-based approach does.

Qadeer *et al.* [19] proposed an algorithm to find single TSCC in the context of *safe replacement* in sequential equivalence checking [21, 22]. In this approach, multiple TSCCs are not considered.

Xie and Beeral proposed a reachability-based algorithm to find all TSCCs iteratively [27]. This is also a symbolic approach that outperforms the method in [13]. However, this approach does not consider initial states.

None of the above previous work on finding TSCCs was used in real designs in practice, due to the design sizes. Our abstraction-based approach is the first in publication to handle large designs in practice.

Case *et al.* [5] proposed a method finding *transient signals* using ternary simulation. A transient signal is a toggle deadlock on over-approximate reachable states. The toggle deadlock checking in this paper finds transients signals in exact reachable states.

3. IMPROVED LIVELOCK CHECKING

3.1 State Classification

The state classification in [27] consists of one transient class and one or more recurrence classes. However, in hardware verification, initial states are given to verify the hardware behavior only in reachable state space. One problem of the state classification in [27] is that there is no distinction between reachable TSCCs and unreachable TSCCs from the initial states. Also, the reachable TSCCs may vary depending on initial states.

We propose a new state classification that is shown in Figure 1, assuming that there is one single initial state. Handling multiple initial states is explained in Section 3.3.

DEFINITION 2. *STSCC is a sink TSCC that has incoming edges from any states outside the TSCC.*

We first define *sink TSCC* (in short, STSCC) in Definition 2. The new state classification consists of main group, transient group, livelock groups (reachable STSCCs) and unreachable TSCCs for a given initial state. The transient class in [27] is further classified into main group or transient group. Main group is an SCC containing the initial state and there exists either one or no main group. The recurrence classes in [27] are further classified into livelock groups (reachable STSCCs) and unreachable TSCCs. When there is no livelock, there exists only one SCC which is the main group.

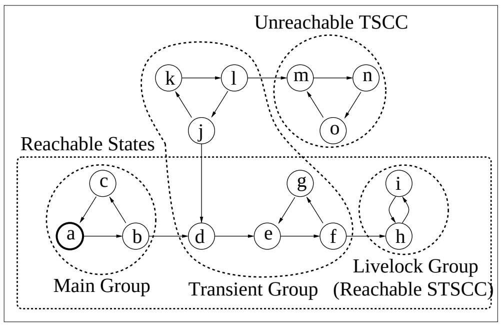

Figure 1: State classification.

In Figure 1, there are states a through o and a is the initial state that is marked with thick circle. Among all states, the reachable states are a through i inside the dotted rectangle. The unreachable states are j through o outside the dotted rectangle. There are five SCCs that are $\{a, b, c\}, \{e, f, g\}, \{h, i\}, \{j, k, l\}, and \{m, n, o\}$. Since a is the initial state, $\{a, b, c\}$ becomes the main group. $\{h, i\}$ and $\{m, n, o\}$ are TSCCs and only $\{h, i\}$ is a livelock group (reachable STSCC) since it is reachable from a. $\{m, n, o\}$ is called an unreachable TSCC. The rest states, $\{d, e, f, g, j, k, l\}$, belong to the transient group in which the states are contained in neither the main group nor the TSCCs.

3.2 Finding Livelock

We first define transition relation in Definition 3 to explain our algorithms to check livelock.

DEFINITION 3. *Let* $x = \{x_1, \ldots, x_n\}$, $y = \{y_1, \ldots, y_n\}$, *and* $w = \{w_1, \ldots, w_p\}$ *be sets of variables ranging over* $B = \{0, 1\}$. *A (finite state) machine is a pair of boolean functions* $\langle Q(x, w, y), I(x)\rangle$, *where* $Q : B^{2n+p} \rightarrow B$ *is 1 if and only if there is a transition from the state encoded by* x *to the state encoded by* y *under the input encoded by* w. $I : B^n \rightarrow B$ *is 1 if the state encoded by* x *is an initial state.* $Q(x, w, y)$ *is called transition relation. The sets* x, y, *and* w *are called the present state, next state, and input variables, respectively.*

The procedure $ComputeForwardSet$ in Figure 2 is a modified version of the procedure $forward\_set$ in [27] in order to compute forward set of a given state s only within the given care states *careSet* in the procedure and to perform *early termination* when *stop* is not ZERO (empty BDD). $\Downarrow$ represents a *restrict operator* [7] that is used to minimize the transition relation with respect to *careSet* in Line 2. The minimized transition relation is denoted by $\tilde{Q}$. In Line 7, $y \leftarrow x$ represents that y variables are replaced by x variables by BDD substitution. Early termination is another big difference from $forward\_set$ in [27] and is used in Figure 3. This is to bail out computing forward set as soon as any newly reached state intersects with the states in *stop* as in Line 11. *BddIteConstant* is a BDD ITE(if-then-else) operation without creating a new BDD node. O is an array of states to store newly reached states at each iteration and O is called *onion rings*. These onion rings are used later in Section 3.3. $ComputeForwardSet$ returns the forward set $F(s)$ and the onion rings O.

```
   ComputeForwardSet($Q$, $careSet$, $s$, $stop$) {
1      $F(s)$ = ZERO;
2      $\tilde{Q}(x, w, y) = Q(x, w, y) \Downarrow careSet$;
3      $frontier(x) = s$;
4      Put $s$ in $O$;
5      while ($frontier(x) \neq$ ZERO) {
6          $image(y) = \exists x, w.\, \tilde{Q}(x, w, y) \wedge frontier(x)$;
7          $image(x) = image(y)|_{y \leftarrow x}$;
8          $F(s) = F(s) \vee image(x)$;
9          $frontier = image(x) \wedge \neg F(s)$;
10         Put $frontier$ in $O$;
11         if (BddIteConstant($frontier$, $stop$, ZERO) != ZERO)
12             break;
13     }
14     return ($F(s)$, $O$);
   }
```

Figure 2: Computing forward set.

$ComputeBackwardSet$ is a dual procedure to $Compute$ $ForwardSet$, except not using *stop* and not computing the onion rings O.

Figure 3 is a procedure for finding TSCCs from the given set of states S. The procedure $FindTSCCs$ is a modified version of the procedure $State\_classification$ in [27]. The modified procedure utilizes care states *careSet*, assuming S

is not necessarily all state space. T is a set of transient states in S, and R is an array of TSCCs in S. *PickOneState* in Line 5 picks a random state from *careSet* as a seed state to find a TSCC. In Line 7, early termination is used in computing the forward set $F(s)$, by setting *stop* in Figure 2 as the negation of $B(s)$. This is because while we compute $F(s)$ within $B(s)$ for the state s, once any state outside $B(s)$ is reachable from s, all states in $B(s)$ are transient. Another big difference is trimming transient states in Line 12 and 16. $TrimTransient(Q, careSet, T, dir)$ trims out the transient states from the current care states by the given direction (dir) that is either PREFIX, SUFFIX, or BOTH. PREFIX(SUFFIX) means to trim out the lasso prefix(suffix) states. This is the same technique used in finding SCCs [20]. Finally, *FindTSCCs* returns R (a set of TSCCs) and T (a set of transient states).

```
   FindTSCCs($Q$, $S$) {
1      $R = \{\ \}$;
2      $T$ = ZERO;
3      $careSet = S$;
4      while ($careSet \neq$ ZERO) {
5          $s$ = PickOneState($careSet$);
6          $B(s)$ = ComputeBackwardSet($Q$, $careSet$, $s$);
7          $F(s)$ = ComputeForwardSet($Q$, $careSet$, $s$, $\neg B(s)$);
8          if ($F(s) \subseteq B(s)$) {
9              $R = R \cup F(s)$;
10             $T = T \vee (B(s) \wedge \neg F(s))$;
11             $careSet = careSet \wedge \neg B(s)$;
12             TrimTransient($Q$, $careSet$, $T$, PREFIX);
13         } else {
14             $T = T \vee (s \vee B(s))$;
15             $careSet = careSet \wedge \neg(s \vee B(s))$;
16             TrimTransient($Q$, $careSet$, $T$, BOTH);
17         }
18     }
19     return $(R, T)$;
   }
```

Figure 3: Finding TSCCs.

```
   FindLivelock($Q$, $S$, $s$) {
1      ($F(s)$, $O$) = ComputeForwardSet($Q$, $S$, $s$, ZERO);
2      $B(s)$ = ComputeBackwardSet($Q$, $S$, $s$);
3      $reached = F(s) \vee s$;
4      if ($F(s) \subseteq B(s)$) {
5          $M = F(s)$;
6          $R = \{\ \}$;
7          $T$ = ZERO;
8      } else {
9          $M = F(s) \wedge B(s)$;
10         $careSet = F(s) \wedge \neg(M \vee s)$;
11         TrimTransient($Q$, $careSet$, $T$, PREFIX);
12         ($R$, $T_R$) = FindTSCCs($Q$, $careSet$);
13         if ($s \notin M$)
14             $T_R = T_R \vee s$;
15         $T_U = B(s) \wedge \neg M$;
16         $T = T_R \vee T_U$;
17     }
18     return $(M, R, T, reached, O)$;
   }
```

Figure 4: Finding livelock.

Figure 4 shows the procedure to perform our new state classification. As explained in Section 3.1, we find main group (M), transient group (T), and livelock groups (R) from the given initial state (s) within the given care states (S). *FindLivelock* starts computing forward set $F(s)$ and backward set $B(s)$ in Line 1 and 2. In Line 3, *reached* is the reached states from s in S. If $F(s) \subseteq B(s)$ in Line 4, there is no livelock in S. In this case, $F(s)$ becomes the main group and both R and T are set to empty in Line 5-7. If $F(s) \nsubseteq B(s)$ in Line 8, there must exist at least one livelock group. In this case, M is computed by intersecting $F(s)$ and $B(s)$ in Line 9. *careSet* is set to a subset of $F(s)$ in Line 10. The lasso prefix states in *careSet* are trimmed out in Line 11. T_R represents the set of transient states that are reachable from s. In Line 12, R and T_R are computed by calling *FindTSCCs* with *careSet*. If $s \notin M$ (means that the main group is empty), s is added to T_R in Line 13-14. T_U represents the set of transient states that are unreachable from s and T_U is computed in Line 15. T is computed by union of T_R and T_U in Line 16.

3.3 Multiple Initial States

It is possible for a design to have multiple initial states when some of the state variables do not have concrete initial values. In the presence of multiple initial states, finding livelock groups has to be devised correctly to avoid false positives and redundant computations.

Figure 5 shows an example with multiple initial states. In this example, there are six states, $S = \{a, b, c, d, e, f\}$. There are two SCCs, $\{a, b, c\}$ and $\{d, e, f\}$. We can see that $\{d, e, f\}$ is a TSCC. a and d are initial states, $I = \{a, d\}$, as shown with thick circles. Suppose that we compute livelock by calling $FindLivelock(Q, S, I)$. Then, we get $F(I) = B(I) = M = \{a, b, c, d, e, f\}$ and $R = \{\}$ which is not correct since there is a reachable TSCC. Now, let us try to call *FindLivelock* for each single initial state. First for the initial state a, we get $F(a) = \{a, b, c, d, e, f\}$ and $B(a) = \{a, b, c\}$. This gives us $M_a = \{a, b, c\}$ and $R_a = \{d, e, f\}$. There is a livelock group R_a for the initial state a. Now, for the initial state d, $F(d) = \{d, e, f\}$ and $B(d) = \{a, b, c, d, e, f\}$. This gives us $M_d = \{d, e, f\}$ and $R_d = \{\}$ and $T_U = \{a, b, c\}$. There is no livelock group for the initial state d. Therefore, we can see that livelock checking has to be applied for each single initial state separately in the presence of multiple initial states.

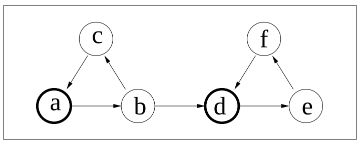

Figure 5: Multiple initial states.

THEOREM 3. *When there are two initial states (i_0 and i_1), if i_1 is included in the reached states from i_0, the livelock groups from i_1 are a subset of the livelock groups from i_0.*

PROOF. Since i_1 is included in the reached states from i_0, i_1 is in either main, transient, or livelock groups from i_0. When i_1 is in the main group, the same livelock groups from i_1 are obtained. When i_1 is in the transient group, all or a subset of the livelock groups i_1 is obtained. When i_1 is in one of the livelock groups, the livelock group including i_1 becomes the main group from i_1, and no livelock group exists from i_1 since the other livelock groups from i_0 become unreachable TSCCs from i_1. From the above three cases, no new livelock group is obtained from i_1 compared to the ones from i_0. Therefore, the livelock groups from i_1 are a subset of the livelock groups from i_0. □

Theorem 3 says that when there is large number of initial states, we can skip livelock checking for any initial states that are already in the forward sets of other initial states. In Figure 5, livelock checking for the initial state d can be skipped because of $d \in F(a)$, assuming that a is used first. However, there is an order dependency on which initial state is used first. If d is used first, we still need to run livelock checking with a. In practice, the number of calls to

FindLivelock is greatly reduced because of Theorem 3 in the presence of multiple initial states.

Figure 6 is the top-level procedure that checks livelock with multiple initial states. *CheckLivelock* takes transition relation(Q), a set of states(S), a set of initial states(I), and a concrete machine(C) as procedure inputs. The use of C is explained in Section 4. *CheckLivelock* first finds livelock groups in the reachable states in Line 1-17 and then it finds TSCCs in the unreachable states in Line 18-23. The *while* loop (Line 6-17) performs livelock checking for a current initial state s until all initial states are covered with iteration index k. For this, *remaining* is initially set to I in Line 3 and updated by eliminating the newly reached states $reached_k$ from *remaining* in Line 13. *reached* is the reached states from all initial states. *reached* is initially set to ZERO in Line 1 and updated by adding $reached_k$ that is the reached states from s in Line 12. Then, the next initial state is chosen from *remaining* in Line 15. T_U is the union of unreachable transient states from each initial state. T_U is initially set to ZERO in Line 2 and updated by adding the unreachable states of T_k in Line 14. For the current initial state s, *FindLivelock* is called in Line 7. $|R_k|$ represents the number of livelock groups in R_k in Line 8. For each R_k^j, a trace $trace_k^j$ is generated in Line 9 and the livelock is reported with the trace in Line 10. Generating trace is explained in Section 4.2 and reporting livelock is explained in Section 4.3.

```
   CheckLivelock($Q$, $S$, $I$, $C$) {
1      $reached$ = ZERO;
2      $T_U$ = ZERO;
3      $remaining = I$;
4      $k = 0$;
5      $s$ = PickOneState($I$);
6      while ($s \neq$ ZERO) {
7          $(M_k, R_k, T_k, reached_k, O_k)$ = FindLivelock($Q$, $S$, $s$);
8          for ($j = 0$; $j < |R_k|$; $j$++) {
9              $trace_k^j$ = GenerateTrace($C$, $R_k^j$, $s$, $O_k$);
10             ReportLivelock($s$, $M_k$, $R_k^j$, $T_k$, $trace_k^j$);
11         }
12         $reached = reached \vee reached_k$;
13         $remaining = remaining \wedge \neg reached_k$;
14         $T_U = T_U \vee (T_k \wedge \neg reached_k)$;
15         $s$ = PickOneState($remaining$);
16         $k$++;
17     }
18     $careSet = \neg(reached \vee T_U)$;
19     if ($careSet \neq$ ZERO) {
20         $R_k$ = FindTSCCs($Q$, $careSet$);
21         for ($j = 0$; $j < |R_k|$; $j$++)
22             ReportUnreachLivelock($R_k^j$);
23     }
   }
```

Figure 6: Checking livelock.

Once all reachable livelock groups are found, we next find unreachable TSCCs. We set the care states *careSet* to the negation of all visited states so far in Line 18, then call *FindTSCCs* with *careSet* in Line 20. If there is any unreachable TSCC, the TSCC is reported in Line 22.

4. LIVELOCK CHECKING ON FSM

To check whether a livelock exists in a design or not, the checking should be done on the whole design. However, this is infeasible due to the size of the design in practice. Thus, we propose a practical method for checking livelock on FSMs on the design.

Even when we check livelock on an FSM, the entire COI logic of the FSM must be considered in order to get an exact result on livelock. However, this is still computationally very expensive or not feasible, in most real designs. Thus, we propose a framework for abstraction-based livelock checking on an abstracted COI of the FSM. Once we find a livelock on the abstract machine, we justify whether the livelock exists on the concrete machine. Notice that a livelock on the abstract machine can be mapped into more than one livelock on the concrete machine.

Figure 7 shows how an abstract machine is obtained from the COI of an FSM. Suppose an FSM that has two state variables f and g. Then, we compute the COI of the FSM. Suppose that there are state variables $\{a, b, c, d, e\}$ in the COI of the FSM. The size of the abstract machine is predefined and let us suppose that the size is N. Then, a set of influential latches from the COI is computed from the FSM variables. The minimum abstract machine is the FSM itself and the maximum abstract machine is the concrete machine. In this example, N=4 and we get the abstract machine $\{f, g, d, e\}$.

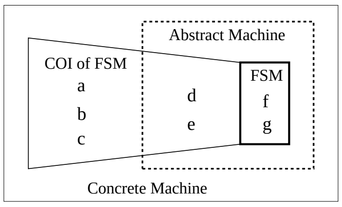

Figure 7: Abstract machine.

THEOREM 4. *If any state in a livelock group on an abstract machine is reachable from the initial state on the concrete machine, the livelock exists on the concrete machine.*

PROOF. Since the abstraction is an over-approximation, the set of all transitions on the abstract machine is a superset of the set of all transitions on the concrete machine. Since there is no path from any state in the livelock group to any state in the main group on the abstract machine, there is still no path from any projected states of the livelock group on the concrete machine to any projected states of the main group on the concrete machine. Now, suppose that the livelock does not exist on the concrete machine. In order for the livelock not to exist on the concrete machine, the only condition is that there is no path from the projected main group to the projected livelock group on the concrete machine. In other words, the projected livelock group has to be unreachable from the initial state. However, this contradicts the assumption that any state of the livelock group is reachable from the initial state on the concrete machine. Therefore, the livelock group still exists on the concrete machine. □

Thanks to Theorem 4, this abstraction-based livelock finds a livelock on small abstract machine using BDD-based symbolic method, then justifies the existence of the livelock on the concrete machine by trace concretization in Section 4.2, by using SAT techniques that can handle large designs. The abstraction-based livelock checking is an incomplete method in the sense that it does not provide the proof of no livelock unless the checking is performed on a concrete machine. No livelock on an abstract machine does not guarantee no livelock on the concrete machine. However, the abstraction-based livelock checking enables finding real livelock errors on industrial large designs.

4.1 Causality Checking

Let V be the set of state variables in an abstract machine for livelock checking. Suppose that $R(V)$ is the reached states in the abstract machine and $L(V)$ is a livelock group containing a TSCC. Also, suppose that v is a state variable in V. We are interested in whether v *contributes* to the livelock as in Definition 4. This is called *variable causality.*

DEFINITION 4. *When a livelock exists in the abstract machine, a variable v in V* contributes *to the livelock if the livelock disappears by eliminating v from the abstract machine. In other words, there is no livelock in another abstract machine that is composed of the variables, $V \backslash v$.*

Equation 1 shows a condition for existence of livelock.

$$L(V) \subset R(V) \tag{1}$$

Now, let $\tilde{R}$ be the quantified reached states and $\tilde{L}$ be the quantified livelock states with respect to a state variable v, as shown in Equation 2 and 3.

$$\tilde{R}(V \backslash v) = \exists_v . R(V) \tag{2}$$

$$\tilde{L}(V \backslash v) = \exists_v . L(V) \tag{3}$$

Then, it is determined by Equation 4 to check whether the variable v contributes to the livelock. Theorem 5 says that if Equation 4 holds, v contributes to the livelock.

$$\tilde{L}(V \backslash v) \subset \tilde{R}(V \backslash v) \tag{4}$$

THEOREM 5. *When a livelock group is found on an abstract machine ($L(V) \subset R(V)$), if $\tilde{L}(V \backslash v) \subset \tilde{R}(V \backslash v)$ holds for a variable v, the variable v contributes to the livelock.*

PROOF. Let M_1 be the machine consisting of V and suppose that a livelock group exists in M_1. Let M_2 be the machine consisting of $(V \backslash v)$ by eliminating v from M_1. Also, let T_1 (T_2) be the set of transitions in M_1 (M_2), respectively. Since M_2 is an over-approximated machine from M_1, M_2 has more transitions than M_1 ($T_1 \subset T_2$). Let T_d be the difference between T_1 and T_2. If there is any transition (in T_d) that makes a path from any state in the livelock to any state in the main group in M_2, the livelock group merges into the main group and both groups become a single SCC, yielding $\tilde{L}(V \backslash v) = \tilde{R}(V \backslash v)$. Thus, M_2 becomes a machine without the livelock. This means that v is a necessary variable to have the livelock in M_1. Therefore, if $\tilde{L}(V \backslash v) = \tilde{R}(V \backslash v)$, v contributes to the livelock. $\square$

This causality checking can also be applied to a set of variables, especially with FSM variables, in order to report whether the livelocks are related with the FSM. Let F be the set of variables in an FSM and C be the set of variables in the COI of the FSM. Suppose that $R(F, C)$ is the reached states in the abstract machine and $L(F, C)$ is a livelock group containing a TSCC. The quantified reached states and the quantified livelock states are computed in Equation 5 and 6 with respect to the FSM variables, respectively.

$$\tilde{R}(C) = \exists_F . R(F, C) \tag{5}$$

$$\tilde{L}(C) = \exists_F . L(F, C) \tag{6}$$

Then, Equation 7 shows the causality checking with the FSM variables to check whether the FSM variables contribute to the livelock.

$$\tilde{L}(C) = \tilde{R}(C) \tag{7}$$

4.2 Trace concretization

Once a livelock group is found on an abstract machine, we need to justify whether the livelock group is reachable on the concrete machine. This can be done by the following three steps. The first step is to pick a target state in the livelock group. The target state is chosen randomly from the livelock group, but is one of the closest states to the initial states by using the onion rings O_k in Figure 6. The second step is to generate an abstract trace. Starting from the target state, an abstract trace can be computed by applying BDD-based pre-image computation iteratively until the initial state is reached. The third step is to generate a concrete trace by making a BMC (Bounded Model Checking [2]) problem from the abstract trace, in order to see whether the livelock group is reachable on the concrete machine. An efficient approach for concretization was proposed in [18]

4.3 Reporting Livelock

Once a concrete trace is generated for a livelock group, the livelock is real on the concrete machine. We report the livelock group with the state classification mentioned in Section 3.1. A livelock group is reported with its initial state, the main group, transient group, and the unreachable states in terms of the number of states and the percentage in each group on the abstract machine.

By looking at the transient and livelock groups, we can see what fraction of the state space is in problematic zone. A good design is expected to have only one main group per one initial state without any transient and livelock groups, unless the design has an intended reset sequence to a normal mode.

5. TOGGLE DEADLOCK CHECKING

There is another important design property, called *toggle deadlock* that is related to livelock. A livelock may occur for multiple state variables of a design, whereas a toggle deadlock may occur on a single state variable. A state variable has a toggle deadlock if the variable initially toggles, but the variable gets stuck at a constant value after a certain number of cycles.

Figure 8 shows an example of toggle deadlock. There are two state variables $\{a, b\}$ and four states $\{s_0, s_1, s_2, s_3\}$ as in the example. Provided that s_0 is the initial state, the main group is $\{s_0, s_1\}$ and the livelock group is $\{s_2, s_3\}$. Once the state transition reaches to s_2 that is a state in the livelock group, the value of b gets stuck at 1, whereas a still toggles. Thus, we say that b has a toggle deadlock.

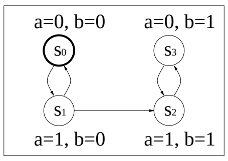

Figure 8: Toggle deadlock.

THEOREM 6. *If there is no STSCC in a design, there is no toggle deadlock on any variable.*

PROOF. To be a toggle deadlock, a variable is supposed to toggle at a cycle and to hold the value forever from the cycle. No STSCC implies that there is only main group in the design. If a variable appears as constant in the main group, the variable is a constant. However, the main group does not have any prefix behavior. This means it is not possible for the variable to get toggled before the main group. Therefore, no STSCC implies no toggle deadlock. $\square$

Theorem 6 shows that toggle deadlock occurs in the presence of a livelock. It is also possible that there is no toggle deadlock on a design that has a livelock. Thus, toggle deadlock on a state variable can be computed by two steps. First,

Design	Statistics					Results										
	L	I	F	T	COI	N	Llk	Dlk	New1			New2			TraceGen	
									Time	Mem	Ops	Time	Mem	Ops	Time	Len
D1	1163	1330	7	-	632	30	0	-	16:20	107.1	44	15:31	107.9	41	-	-
						60	0	-	41:11	136.7	66	41:31	137.7	66	-	-
						90	0	-	2:30:33	224.1	81	2:24:50	224.3	81	-	-
						120	-	-	time-out (> 24h)			time-out (> 24h)			-	-
D2	385	352	25	-	68	30	0	-	0:01	21.0	12	0:01	21.1	12	-	-
						60	0	-	0:40	38.7	28	0:40	38.7	28	-	-
						68	12032	-	4:47:52	95.7	386439	0:53:58	96.5	70329	-	-
D3-F1	32541	912	2	-	28941	30	1	-	0:49	140.4	54	0:49	140.5	55	6:37	66
D3-F2	32541	912	4	-	28930	4	1	-	0:09	129.1	9	0:09	129.1	12	1:54	14
			-	4	28930	30	-	1	1:31	132.8	168	1:31	132.8	172	2:11	14

Table 1: Experimental results.

we find STSCCs on an abstract machine from the state variable. The abstract machine is made in the same way as in livelock checking on FSM. Secondly, we evaluate the value of the state variable in the livelock if the livelock exists.

Figure 9 shows the procedure that checks toggle deadlock on a given state variable t. *CheckToggleDeadlock* takes transition relation (Q), a set of states (S), a set of initial states (I), a concrete machine (C), and the state variable (t) as procedure inputs. *CheckToggleDeadlock* is similar to *CheckLivelock* in Figure 6. For each reachable livelock group R_k^j in Line 6, *TestToggleDeadlock* checks whether the value of t toggles or not in the livelock group and returns *dlk* and c in Line 7. *dlk* represents whether the state variable is in toggle deadlock or not, and c is the constant value (0 or 1) in the case of toggle deadlock.

```
   CheckToggleDeadlock($Q$, $S$, $I$, $C$, $t$) {
1      $remaining = I$;
2      $k = 0$;
3      $s$ = PickOneState($I$);
4      while ($s \neq$ ZERO) {
5          $(R_k, reached_k, O_k)$ = FindLivelock($Q$, $S$, $s$);
6          for ($j = 0$; $j < |R_k|$; $j$++) {
7              ($dlk$, $c$) = TestToggleDeadlock($R_k^j$, $t$);
8              if ($dlk$) {
9                  $trace_k^j$ = GenerateTrace($C$, $R_k^j$, $s$, $O_k$);
10                 ReportToggleDeadlock($s$, $R_k^j$, $trace_k^j$, $c$);
11             }
12         }
13         $remaining = remaining \wedge \neg reached_k$;
14         $s$ = PickOneState($remaining$);
15         $k$++;
16     }
   }
```

Figure 9: Checking toggle deadlock.

6. EXPERIMENTAL RESULTS

We have implemented the proposed livelock checking and toggle deadlock checking algorithms. Table 1 shows our experimental results on livelock and toggle deadlock checking, generated on a 1.4 GHz Intel processor machine with 4 GB memory running Red Hat Linux.

The first column lists the design names. The next five columns present the statistics on the designs, in terms of the number of latches (L), the number of inputs (I), the number of latches in FSM (F), the number of toggle signals to check (T), and the number of latches in the COI of either FSM and a toggle signal (COI). The next three columns show the results on livelock and toggle deadlock checking. The column with N shows how many latches were in the abstract machine. The column with *Llk* shows how many livelock groups are found and the column with *Dlk* shows how many toggle deadlock are found. The next six columns compare the performance between two methods ($New1$ and $New2$), in terms of time(*Time*), memory(*Mem*), and the number of image/pre-image computations(*Ops*). $New1$ is the proposed method without the trimming technique, whereas $New2$ is the proposed method with the trimming technique. The times are in the form of *hh:mm:ss* and the memory consumptions are in M-byte. The final two columns(*TraceGen*) show the results on trace generation on concrete machine for the livelock or toggle deadlock found by $New2$, and *Time* shows the time spent for trace generation and *Len* shows the trace length.

We have chosen 3 industrial designs ($D1$, $D2$, and $D3$). For each design, we have run livelock or toggle deadlock checking on several sizes of abstract machines with the multiples of 30 latches. We have set the maximum run time to 24 CPU hours.

In $D1$, there is one FSM automatically extracted. The FSM consists of 7 latches and contains 632 latches in its COI. We can see that the run time is exponentially increased, depending on the size of the abstract machine. On this design, the livelock checking became infeasible when N=120.

In $D2$, there is also one FSM that was user-specified. The FSM consists of 25 latches and contains only 68 latches in its COI. This design has a livelock group. However, the livelock was not detected when N=30 and N=60. The livelock was detected only when all the latches in the COI were included in the abstract machine. In other words, the abstract machine is the concrete machine at the FSM point of view. Since the livelock was found on the concrete machine, trace concretization is not required since the abstract trace in Section 4.2 is already a concrete trace.

$D2$ is the only design showing a significant performance difference between $New1$ and $New2$ in the table. This is because this design has many transient states as well as many livelock groups. In this case, the trimming technique significantly reduced the number of image/pre-image operations from 386K to 70K (5.5X reduction) that gave big speed-up from 5 hours to 1 hour (5X speed-up). This shows that the trimming technique helps the performance when there are many transient states. When there is no transient states, the trimming technique becomes a pure overhead as shown in $D3$. However, the overhead is almost negligible from the experiment.

In $D3$, there are two FSMs ($F1$ and $F2$). $F1$ is composed of 2 latches and a livelock was found with N=30 within 49 seconds. The livelock was justified by trace concretization that took 397 seconds, and the trace length was 66. $F2$ is composed of 4 latches and a livelock was found with N=4 (the FSM itself) in 9 seconds. The livelock was also justified by trace concretization that took 114 seconds, and the trace length was 14. We have also tried the toggle deadlock checking on $F2$ separately from the livelock checking. A toggle deadlock was found in 90 seconds and the concrete trace was generated in 131 seconds. $D3$ shows the value of abstraction-based livelock and toggle deadlock checking.

Table 2 shows a comparison on finding all SCCs with four algorithms (XB [28], Lockstep [3], Skeleton [12], IXB [20]) on the design $D2$ from Table 1. In this design, the number of recurrent states is 2.07e8 and the number of transient states is 1.2e6 that is only 0.6% of all states. However, it turned out that how to handle these transient states efficiently is the key factor in the performance. One main difference between XB and IXB is that IXB trims out those transient states as much as possible. This trimming technique makes the IXB method outperform on this design: faster in time (more than 15X) and fewer number of image operations (more than 10X) than the other methods. This explains why $New2$ outperformed on $D2$ in Table 1. Table 2 also shows why livelock checking is done by finding TSCCs instead of SCCs. Finding all livelock groups took 54 minutes, whereas finding all SCCs took 100 minutes (2X) even with IXB.

Method	Time	Memory	Ops	SCCs	States
XB	84:07:56	98.2	1013333	19458	2.08e8
Lockstep	45:55:53	237.3	2590724		
Skeleton	26:01:54	266.5	2609008		
IXB	1:39:47	92.5	102990		

Table 2: Finding all SCCs in D2.

7. CONCLUSIONS

We have presented a framework for abstraction-based livelock and toggle deadlock checking, in order to handle large designs in practice. Since exact livelock and toggle deadlock checking is infeasible on real designs directly, our approach is to check livelock and toggle deadlock on abstract machine of either an FSM or a toggle signal. Once we find a livelock or toggle deadlock, we justify the livelock or toggle deadlock on the concrete machine by concretizing the abstract trace on the concrete machine.

Even though the proposed approach does not prove the non-existence of livelock or toggle deadlock on a design unless the design is small enough to handle, this approach finds livelocks or toggle deadlocks on the design if there exists.

To the best of our knowledge, it is the first approach to use the abstraction-based livelock checking and also the first approach for checking toggle deadlock. The experimental results showed that the abstraction-based approach finds livelock errors on the real designs.

As future work, we are interested in improving the concretization, finding more accurate influential latches, and optimizing the computations with multiple FSMs or toggle signals by considering the overlaps in their COIs.

8. REFERENCES

[1] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *International Workshop in Formal Methods for Industrial Critical Systems*, pages 160–177, 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[3] R. Bloem, H. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Formal Methods in Computer Aided Design*, pages 37–54, 2000.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[5] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman. Enhanced verification by temporal decomposition. In *Formal Methods in Computer Aided Design*, pages 37–54, 2009.

[6] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate fsm traversal based on circuit analysis. *IEEE Transactions on Computer-Aided Design*, 15(12):1451–1464, Dec. 1996.

[7] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, Nov. 1990.

[8] O. G. E. M. Clarke and D. Peled. *Model Checking.* The MIT Press, 1999.

[9] N. Een and N. Sorensson. *MiniSat.* http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat.

[10] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[11] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[12] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582, 2003.

[13] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer-Aided Design*, 15(12):1479–1493, Dec. 1996.

[14] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.

[15] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1994.

[16] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.

[17] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proceedings of the Design Automation Conference*, pages 260–265, June 1993.

[18] K. Nanshi and F. Somenzi. Constraints in one-to-many concretization for abstraction refinement. In *Proceedings of the Design Automation Conference*, pages 569–574, 2009.

[19] S. Qadeer, R. K. Brayton, V. Singhal, and C. Pixley. Latch redundancy removal without global reset. In *Proceedings of the International Conference on Computer Design*, pages 432–439, 1996.

[20] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, Nov. 2000. LNCS 1954.

[21] V. Singhal. *Design replacements for sequential circuits.* Ph.D. dissertation, University of California at Berkeley, 1996.

[22] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton. Theory of safe replacements for sequential circuits. *IEEE Transactions on Computer-Aided Design*, 20(2):249–265, Feb. 2001.

[23] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[24] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, Apr. 1995.

[25] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[26] T.-H. Wang and T. Edsall. Practical FSM analysis for verilog. In *IVC-VIUF '98: Proceedings of the International Verilog HDL Conference and VHDL International Users Forum*, pages 52–58, 1998.

[27] A. Xie and P. A. Beeral. Efficient state classification of finite-state markov chains. *IEEE Transactions on Computer-Aided Design*, 17(12):1334–1339, Dec. 1998.

[28] A. Xie and P. A. Beeral. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design*, 19(10):1225–1230, Oct. 2000.