Decision Ordering Based Property Decomposition for Functional Test Generation

Mingsong Chen
mschen@sei.ecnu.edu.cn
Software Engineering Institute
East China Normal University, China

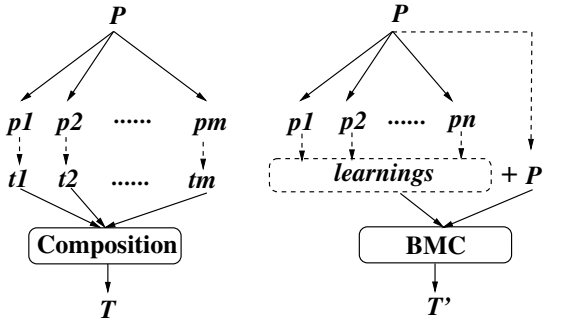
Prabhat Mishra
prabhat@cise.ufl.edu
CISE Department
University of Florida, USA

Abstract

SAT-based BMC is promising for directed test generation since it can locate the reason of an error within a small bound. However, due to the state space explosion problem, BMC cannot handle complex designs and properties. Although various optimization methods are proposed to address a single complex property, the test generation process cannot be fully automated. This paper presents an efficient automated approach that can scale down the falsification complexity using property decomposition and learning techniques. Our experimental results using both software and hardware benchmarks demonstrate that our approach can drastically reduce the overall test generation effort.

I. Introduction

Satisfiability (SAT) based Bounded Model Checking (BMC) [1] is widely used for directed test generation. However, when checking a large design with complex properties (i.e., properties with large cone of influence or deep bounds), BMC based methods are very costly since large SAT instances indicate long SAT search time.



a) Test−oriented decomposition b) Our learning−oriented decomposition

Fig. 1. Two property decomposition techniques

To address this problem, Koo et al. proposed a property decomposition technique [2] as shown in Figure 1a. The basic idea is to decompose a complex property into several simple sub-properties, and then compose the tests corresponding to sub-properties to obtain a test for the original property. Since the test generation time of sub-properties is typically several orders of magnitude smaller than the original property, the state space explosion problem can be avoided in many scenarios. However, the composition of tests of sub-properties is a major bottleneck in this method. The human intervention and expert knowledge may be required to use this method. As an alternative, in this paper, we propose a learning-oriented decomposition technique shown in Figure 1b which can be fully automated. Unlike the test-oriented method in [2], our approach is based on the decision ordering learned during the test generation of decomposed sub-properties. Such learnings can be used to drastically accelerate the original property falsification [3]. Therefore the overall test generation effort can be significantly reduced. Our method makes three important contributions: i) it proposes a method to spatially or temporally decompose a complex property into several simple but profitable sub-properties; ii) it proposes an approach to derive learnings from the decomposed sub-properties; and iii) it proposes a method to guide the complex property checking using derived learnings.

The rest of the paper is organized as follows. Section II presents related work on decomposition techniques as well as optimization heuristics for SAT-based BMC. Section III presents our test generation methodology using property decomposition and learning techniques. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Related Work

To address the complexity during the design and verification, various decomposition techniques are proposed. Lin et al. [4] proposed a new formulation of Ashenhurst decomposition based on SAT solving. It can efficiently decompose a Boolean function into a network of smaller sub-functions. However, this method is mainly used in logic synthesis. As described in Section I, although Koo et al. [2] presented a promising design and property decomposition method for test generation, it is hard to automate the composition of generated partial tests corresponding to the decomposed properties.

Sharing learnings across properties can improve overall performance since the repeated validation efforts can be avoided. In [5], Chen and Mishra noticed that when checking a large set of relevant properties, SAT instances of similar properties have a large overlap of CNF clauses and can be clustered. A large number of conflict clauses generated by a base property can be forwarded to other properties in the cluster. As an alternative of conflict clause forwarding, decision ordering heuristics [6] can be used as another learning to improve the SAT searching. In [7], Strichman presented a BMC optimization based on decision ordering by exploiting the characteristics of BMC

This work was partially supported by NSF CAREER award 0746261, 973 Project 2009CB320702, and NSF of China 61021004.

978-3-9810801-7-9/DATE11/ © 2011 EDAA

formulas. When checking a set of similar properties, Chen et al. [3] tuned the decision ordering of the current property based on the decision ordering results of the previously checked properties. By sharing learnings among properties, the overall test generation time can be reduced. However, these learning techniques do not consider how to actively learn from other simpler properties. Moreover, checking the first property in such methods is a major bottleneck because there is no knowledge that can be learned.

To the best of our knowledge, our approach is the first attempt to use decision ordering based learning in property decomposition to enable automated test generation.

III. Test Generation using Property Decomposition and Learning Techniques

This paper focuses on efficient falsification of safety Linear Temporal Logic (LTL [8]) properties. The basic idea is to utilize the learnings from simple properties for complex property checking. The following sections describe each step in detail.

A. Learning-Oriented Property Decomposition

To reduce the complexity during the test generation, this section presents two beneficial decomposition techniques: spatial and temporal property decompositions.

1) Spatial Property Decomposition: For a complex property which involves multiple components of the design, it can be partitioned into multiple component level sub-formulas. For example, a complex system level property P can be broken into 2 sub-properties P_1 and P_2 with different Cone of Influence (COI). Assuming that P_1 has a smaller COI than P, it usually needs less time and space than that of checking the complex property P. If the partial counterexample generated by P_1 can be refined to guide the complex property falsification, the original property is *spatially decomposable*.

Definition 1: A false property P in conjunctive form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or in disjunctive form $p_1 \vee p_2 \vee \ldots \vee p_n$ is *spatially decomposable* if all of the following conditions are satisfied.

- If the decomposed properties are in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, then at least one property p_i $(1 \leq i \leq n)$ has a counterexample. In this case, the bound of P is the minimum bound of p_i which has a counterexample.
- If the decomposed properties are in the form $p_1 \vee p_2 \vee \ldots \vee p_n$, then each property p_i $(1 \leq i \leq n)$ has a counterexample. In this case, the bound of P is the maximum bound of all decomposed properties.
- The counterexamples generated from properties p_i $(1 \leq i \leq n)$ can guide the test generation for property P. ■

According to Definition 1, the following rules can be used for complex property decomposition.

$$\begin{aligned} \neg X(p \vee q) &\equiv \neg X(p) \wedge \neg X(q) \\ \neg X(p \wedge q) &\equiv \neg X(p) \vee \neg X(q) \\ \neg F(p \vee q) &\equiv \neg F(p) \wedge \neg F(q) \end{aligned} \tag{1}$$

The property in the form of $\neg F(p \wedge q)$ and $\neg F(p \rightarrow q)$ cannot be directly decomposed into conjunctive or disjunctive form. However, by introducing a clock *clk* for synchronization, they can be spatially decomposed. It is important to note that the value of the *clk* indicates the bound of the false property. The Equation (2) shows that the counterexample of $\neg F(p \wedge q \wedge clk = k)$ can be refined by the counterexamples of $\neg F(p \wedge clk = k)$ and $\neg F(q \wedge clk = k)$.

$$\begin{aligned} &\neg F(p \wedge q \wedge clk = k) \equiv \neg F(p \wedge clk = k) \vee \neg F(q \wedge clk = k) \\ &\text{where } \neg F(p \wedge q \wedge clk = k) \text{ is false.} \end{aligned} \tag{2}$$

For a property in the form $F(p \rightarrow q)$, p describes the pre-condition and q indicates the post-condition. When $G(\neg p)$ holds, $F(p \rightarrow q)$ will be vacuously true, and the checking of $\neg F(p \rightarrow q)$ will report a counterexample without satisfying the precondition p. This counterexample may not match the original intention. Equation (3) shows that the properties in the form of $\neg F(p \rightarrow q \wedge clk = k)$ can be transformed to $\neg F(p \wedge q \wedge clk = k)$ for test generation. The spatial decomposition in Equation (2) and Equation (3) are similar.

$$\begin{aligned} \neg F(p \rightarrow q \wedge clk = k) &\equiv \neg F(p \wedge q \wedge clk = k) \\ &\equiv \neg F(p \wedge clk = k) \vee \neg F(q \wedge clk = k) \\ &\text{where } \neg F(p \rightarrow q \wedge clk = k) \text{ and } \neg F(p \wedge clk = k) \text{ are false.} \end{aligned} \tag{3}$$

In Equations (1) - (3), we presented several kinds of widely used properties which can be spatially decomposed. In fact, if a complex property can be decomposed in conjunctive form, we need to sort the sub-properties according to their bounds, and check them from the smallest bounds to largest bounds. The counterexample of first falsified property can be used as a counterexample for the complex property.

When checking a complex property which can be decomposed in disjunctive form, it is not necessary to check all its sub-properties. If the COI of a sub-property is similar to the original property, the complexity of such sub-property will be similar to the complex property. In this case, it is not economical to use learning. Therefore we need to figure out sub-properties with smaller COI than the complex property.

According to the commutative law and associative law, for a complex property, we can classify its atomic sub-properties into several clusters. For example, in Equation (4), p_i and p_k are clustered together, and p_j belongs to another cluster.

$$p_i \vee p_j \vee p_k = (p_i \vee p_k) \vee p_j \tag{4}$$

For each cluster, we generate a *refined property* which represents all the atomic sub-properties in the cluster to derive the learning. Based on our experience, the following clustering rules work well for most of the time: 1) in each cluster, all the variables in the sub-formulas should come from the same component (e.g., fetch module in a processor design); 2) in each cluster, all the sub-formulas should describe the related functional scenarios (e.g., fetching instructions and/or data in a processor design).

Algorithm 1 outlines our spatial decomposition method which can derive a set of refined sub-properties with small COI for learning. The inputs of the algorithm are a design model D and a complex property P in disjunctive form. Step 1 initializes the *SD_props* with an empty set. Step 2 tunes sub-properties' order according to the commutative law and clusters sub-properties using the similarity rules. Step 3 selects

the i^{th} cluster. If the COI of such cluster is smaller than $\frac{k}{n}$ of $P's$ COI, step 4 will generate a new refined property *newP* for the i^{th} cluster. Step 5 adds *newP* to *SD_props*. The refined property *newP* for learning represents a cluster of sub-properties as shown in step 3. Finally this algorithm will return a set of refined sub-properties for deriving learnings (described in Section III-B). Since the COI of a refined property in *SD_props* is small, its test generation time will be much smaller than that of the original complex property. It is important to note that this algorithm may return an empty set which means that it is not beneficial or not possible to spatially decompose the complex property.

Algorithm 1: Spatial Decomposition

Input: i) The design model, D
ii) A property P in the form $p_1 \vee p_2 \vee \ldots \vee p_n$
Output: A set of refined sub-properties for learning, *SD_props*
1. *SD_props* = {};
2. $(cluster_1, \ldots, cluster_m) = clustering(P, modular/functional)$;
for *i is from 1 to m* **do**
  3. $cluster\_i = \{prop_1, \ldots, prop_k\}$;
  if $COI(cluster_i) \leq \frac{k}{n} COI(P)$ **then**
    4. generate a refined property *newP* for the $cluster_i$;
    5. $SD\_props = SD\_props \bigcup newP$;
  end
end
return *SD_props*;

2) Temporal Property Decomposition: To eclipse the bound effect, the basic idea of our method is to deduce a long bound property from a sequence of short bound properties. For example, $P1$, $P2$ and $P3$ ($P3 = P$) are properties indicating three different stages of property P. The bound of them are $K1$, $K2$ and $K3$, respectively, and $K1 < K2 < K3$. Because $P1$'s counterexample is similar to the prefix of the $P2$'s counterexample, $P1$'s counterexample contains rich knowledge that can be used when checking $P2$. Similarly, during the property checking, $P3$ can benefit from $P2$. Therefore we can quickly obtain the counterexample (test) for property P. If the counterexamples of lower bound properties can be used to reason about P, the property P is *temporally decomposable*.

Definition 2: A false safety property P is *temporally decomposable* if all the following conditions are satisfied.

- P can be divided into false properties p_1, p_2, ... and p_n ($P = p_n$) with increasing bounds.
- $\neg p_i \rightarrow \neg p_{i+1}$ $(1 \leq i \leq k_n - 1)$, which indicates that the counterexample generated from properties p_i can guide the test generation for property p_{i+1}. ■

In temporal decomposition, finding the implication relation ("$\rightarrow$") between properties is a key process. In our framework, we construct such implication relations by exploring the order between events, which are described by properties indicating different stages of the execution. Generally a system behavior consists of a sequence of strongly relevant *events*. For example, in Figure 2, there are 9 events. We classify the relation between these events in two categories. The *cause effect* relation (marked by $\Rightarrow$) defines the relation of continuous events. For example, if $e1$ happens, then $e2$ should happen in next stage. The *happen before* relation (marked by $\prec$) specifies the relation of conditional events. It indicates which events may happen before other events under some condition. For example, $e2 \prec e3$ means $e2$ may happen before $e3$.

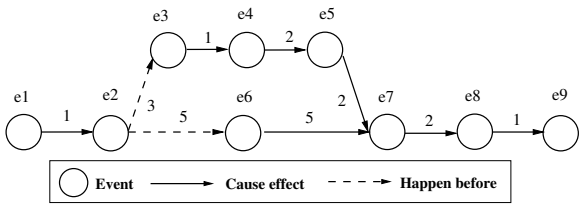


Fig. 2. A DAG of event relation

During test generation, we apply properties in the form of $\neg F(e)$ to indicate that the event e cannot be activated. According to definition 2, the "$\Rightarrow$" relation can be used to derive helpful learnings. For example, in Figure 2, let property $P_1 = \neg F(e_1)$ and property $P_2 = \neg F(e_2)$. Since $e_1 \Rightarrow e_2$ implies $F(e_1) \rightarrow F(e_2)$, i.e., $\neg P_1 \rightarrow \neg P_2$, it shows that $P1$'s counterexample will be helpful for deriving $P2$'s counterexample. Such information can be used as a learning. The "$\prec$" relation also can be used to indicate the learning information. Assuming $e_2 \prec e_3$, the counterexample of $\neg F(e_2)$ is shorter than the counterexample of $\neg F(e_3)$. However, counterexample of $\neg F(e_2)$ may have a large overlap of variable assignments with the counterexample of $\neg F(e_3)$. Therefore the learning from $\neg F(e_2)$ can benefit the test generation of $\neg F(e_3)$.

When checking a large bound property for a transaction, there may be many events along the path to target events. Checking all these events to obtain learnings is time-consuming. For example, assuming that we want to check the property $\neg F(e_9)$, the relation between events is described using a directed acyclic graph (DAG) shown in Figure 2. Each node indicates an event, and each directed edge indicates the relation of "$\Rightarrow$" or "$\prec$", and each edge is associated with the delay between events. In this DAG, there are 8 events that happen before $e9$. However, it is not necessary to check all of them. Since the branch nodes of a DAG contain critical variable assignment information, in our method, we only consider the events which determine the branches along the path from initial state $e1$ to the target state $e9$.

Algorithm 2: Temporal Decomposition

Input: i) An event DAG, D
ii) Initial event *src*, target event *dest*
Output: A property sequence TD_props
1. *path* = Dijkstra(D, *src*, *dest*) to find the shortest delay path;
2. TD_props = (property for *src*);
for *i is from 2 to len (number of events in path)* **do**
  3. $(e_{i-1}, e_i) = (i-1)^{th}$ edge of *path*;
  if $out\_degree(e_{i-1})$ + $in\_degree(e_i) > 2$ **then**
    4. Append the property for e_i to TD_events;
  end
end
return TD_props;

Algorithm 2 describes how to obtain a sequence of properties based on temporal decomposition. It accepts an event DAG with the initial and target events as inputs. Step 1

uses Dijkstra's algorithm [9] to find a shortest *path*. Step 2 initializes the sequence *TD_props* with a property for the initial event. Step 3 and 4 select the *branch events* and append their corresponding properties to the *TD_props*. Finally the algorithm reports the property sequence for deriving learnings. By using this algorithm, $(\neg F(e1), \neg F(e3), \neg F(e7))$ is a property sequence from the temporal decomposition in Figure 2.

B. Decision Ordering Based Learning Techniques

SAT based BMC encodes a property checking problem into a SAT instance (a Boolean formula). A counterexample of the property is a satisfiable variable assignment for this formula. Although the variable assignment of counterexamples derived from the decomposed sub-properties may not satisfy the SAT instance of the complex property, it has a large overlap with the complex property on the variable assignment. Such information can be used as a learning to bias the decision ordering when checking the complex property.

During the SAT search, decision ordering plays an important role to quickly find a satisfiable assignment. Our learning approach is based on Variable State Independent Decaying Sum (VSIDS) method [12]. A major difference is that our method incorporates the statistics of decomposed properties. Since different sub-properties have different bounds, we consider such information in our heuristics.

Let *bounds* be an array which stores the bound of k sub-properties. Because in spatial method the decomposed sub-properties can be independent, the learning between sub-properties is not significant. So we set $bound[i] = 1 (1 \leq i \leq k)$. However for temporal decomposition, the *vstat* information of lower bound properties can further benefit the larger bound property checking. Moreover, the larger bound sub-property is closer to the final properties than smaller bound sub-properties. Therefore, for temporal decomposition based method, the sub-properties are sorted according to the increasing *bound*, and $bound[i]$ indicates the bound of i^{th} property. Let $vstat[sz][2]$ (sz is the maximum Boolean variable index during the complex property checking) be a 2-dimensional array to record the statistics of variable assignments. Initially, $vstat[i][0] = vstat[i][1] = 0$ $(0 < i \leq sz)$. *vstat* will be updated after checking each sub-property. When checking the sub-property p_j, if a variable v_i is evaluated and its value in the counterexample is 0 (false), $vstat[i][0]$ will be increased by $bounds[j]$; otherwise if $v_i = 1$ (true), $vstat[i][1]$ will be increased by $bounds[j]$.

Assuming l_i is a literal of the variable v_i (v_i has two literals, v_i and v_i'), we use $score(l_i)$ to indicate its decision ordering. Initially, $score(l_i)$ is equal to the literal count of l_i. However, at the beginning of SAT searching and periodic score decaying, the literal score will be recalculated. Let

$$bias = \frac{MAX(vstat(v_i), vstat(v_i')) + 1}{MIN(vstat(v_i), vstat(v_i')) + 1}$$

indicate the variable assignment variance. And let

$$score(l_i) = \begin{cases} max(v_i) * bias & (vstat[i][1] > vstat[i][0] \& l_i = v_i) \\ & or (vstat[i][1] < vstat[i][0] \& l_i = v_i') \\ score(l_i) & otherwise \end{cases}$$

The new literal score will be updated using the above formula where $max(v_i) = MAX(score(v_i), score(v_i')) + 1$.

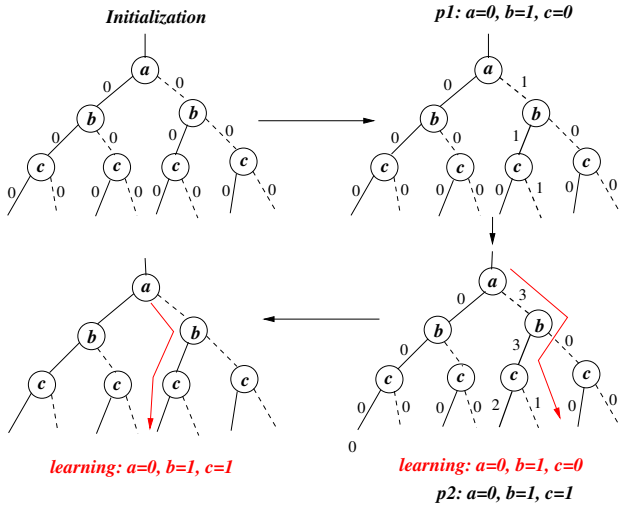


Fig. 3. Learning statistics applied on decision trees

Figure 3 shows an example of temporal decomposition using our heuristic. The complex property P is decomposed into three properties p_1, p_2 and $p_3 (= P)$ with bound 1, 2 and 3 respectively and we assume that we always check the variables in the order of a, b, c. Initially, when checking p_1, there is no learning information. However, after checking p_1, we can predict the decision ordering for p_2 based on the collected *vstat* information from p_1. Also we can predict the decision ordering of $p_3 (= P)$ from the *vstat* of p_1 and p_2. When checking P, the content of *vstat* indicates that variables a is more likely to be 0, b and c are more likely to be 1.

C. Test Generation using Our Method

In this paper, we assume that the bound of complex property and decomposed properties can be pre-determined. Determination of bound is hard in general. However, for directed test generation, the bound can be determined by exploiting the structure of the design. An example of bound determination is presented in Section III-D.

Algorithm 3 describes our test generation methodology. The inputs of the algorithm are a formal model of the design, a set of decomposed properties *props* and their satisfiable bounds *bounds*, and the complex property P with its satisfiable bound $bound_p$. Step 1 generates CNF files in the DIMACS format [10] for each decomposed property in *props*. Step 2 sorts the CNFs by their DIMACS file size. Step 3 initializes *vstat* which is used to keep statistics of the variable assignments for decomposed sub-properties. Then for each decomposed sub-property, we collect its counterexample assignments from step 4 to step 5. For each iteration, we need to update *vstat* statistics. In step 6 and step 7, the complex property P is checked using the decision ordering derived from the decomposed sub-properties. Finally, the algorithm reports a test for the complex property P.

Algorithm 3: Our Test Generation Method

Input: i) Formal model of the design, D
ii) Decomposed properties *props* and satisfiable *bounds*
iii) The property P and satisfiable bound $bound_P$
Output: A test $test_P$ for P
1. $CNFs = BMC(D, props, bounds)$;
2. $(CNF_1, \ldots, CNF_n)$ = sort $CNFs$ using increasing file size;
3. Initialize *vstat*;
for *i is from 1 to the n* **do**
  4. $test_i$ = SAT(CNF_i, *vstat*);
  5. $Update(vstat, test_i, bounds[i])$;
end
6. Generate $CNF = BMC(D,\ P,\ bound_P)$;
7. $test_P$ = SAT(CNF, *vstat*);
return $test_P$;

D. An Illustrative Example

This section presents an example of how to apply decomposition methods on the graph model of a MIPS processor [5]. It consists of five pipeline stages: fetch, decode, execute, memory and writeback. It has four pipeline paths in the execute stage: ALU for integer ALU operation, FADD for floating-point addition operation, MUL for multiply operation and DIV for divide operation. Assume that we want to check a complex scenario that the units *MUL*5 and *FADD*3 will be active at the same time. We generate the property $P = !F(MUL5\_active = 1\ \&\ FADD3\_active = 1)$ which is a negation of the desired behavior. The remainder of this section describes how to generate a test using spatial and temporal decomposition methods respectively.

1) Spatial Decomposition: In the MIPS design, each functional unit has a delay of one clock cycle. To trigger the functional unit *MUL*5, we need at least 7 clock cycles (there are 7 units along the path *Fectch* $\rightarrow$ *Decode* $\rightarrow \ldots \rightarrow$ MUL5). Similarly, to trigger the functional unit *FADD*3, we need at least 5 clock cycles, plus one clock cycle for initialization, we need a total 8 clock cycles for triggering this interaction. Thus the bound of this property is 8. According to Equation (2) and Algorithm 1, property P can be spatially decomposed into two sub-properties as follows, assuming the COI of $P1$ and $P2$ are both smaller than half of COI of P.

```
/* Modified original complex property P' */
P': !F(MUL5_active=1 & FADD3_active=1 & clk=8)
/* Spatially decomposed properties */
P1: !F(MUL5_active=1 & clk=8)
P2: !F(FADD3_active=1 & clk=8)
```

When checking $P1$ and $P2$ individually, we can get the following two counterexamples.

```
   Counterexamples for P1 and P2
Cycles  P1's Inst.        P2's Inst.
  1     NOP               NOP
  2     MUL R2, R2, R0    NOP
  3     NOP               NOP
  4     NOP               FADD R1, R1, R0
```

However, according to Algorithm 3, the test generation for $P2$ is under the guidance of $P1$'s result. Thus, the counterexample of $P2$ guided by $P1$ contains $P1$'s partial behavior (see clock cycle 2 below). So the score of literals which have repetitive occurrences is enhanced.

```
Counterexample for P2 guided by P1
Cycles   P2's Inst. after learning
  1         NOP
  2         MUL R2, R2, R0
  3         NOP
  4         FADD R1, R1, R0
```

The statistics saved in *vstat* indicates an assignment which has a large overlap of the assignments with the counterexample that can activate property P. Thus it can be used as the decision ordering learning to guide the property checking of P.

2) Temporal Decomposition: Temporal decomposition requires figuring out event relation first. Since we want to check property $!F(MUL5\_active = 1\ \&\ FADD3\_active = 1)$, the target event is $MUL5\_active = 1\ \&\ FADD3\_active = 1$. Figure 4 shows the event implications. There are 7 events in this graph, and e_7 is the target event.

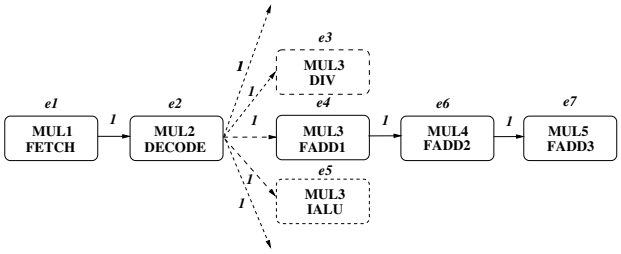


Fig. 4. Event implication graph for property P

Assuming $e1$ is the initial event, from $e1$ to $e7$, there is only one path $e_1 \rightarrow e_2 \rightarrow e_4 \rightarrow e_6 \rightarrow e_7$. Along this path there is a branch node e_2. According to the Algorithm 2, we need to check two events e_1 and e_4 using following properties. By using our learning technique, during the test generation, *P_e4* can benefit from *P_e1*, and P can benefit from *P_e4*. i

```
/* Spatially decomposed properties*/
P_e1: !F(FETCH_active=1 & MUL1_active=1)
P_e4: !F(MUL3_active=1 & FADD1_active=1)
```

IV. Experiments

For test generation, we used NuSMV [11] to derive the CNF clauses (in DIMACS format) and integrated our proposed methods in the state-of-the-art SAT solver zChaff [10] . The experimental results are obtained on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM.

A. A VLIW MIPS Processor

This section presents the experimental results using the same design illustrated in Section III-D. For the MIPS design, we are focusing on the test generation of interaction faults. We generated the properties in the form of $!F(p_1 \& p_2 \& \ldots \& p_n)$ which indicate whether n pipelined components p_i $(1 \leq i \leq n)$ can be activated at the same time. For example, the property $!F(MUL6\_active = 1 \& FADD3\_active = 1 \& DIV\_active = 1)$ asserts that there is no instruction sequence which can activate the components *MUL*6, *FADD*3 and *DIV* at the same time.

We generated 20 properties based on various interaction faults. Since it is hard to figure out the temporal relation between events, 6 properties of them cannot be handled by temporal decomposition. Table I shows the test generation results using our spatial decomposition approach for such

properties. The first column indicates the selected properties. The second column gives the test generation time using zChaff. The third and fourth columns present the number of sub-property clusters and the number of refined sub-properties for deriving learnings. The last two columns show test generation time using our spatial decomposition method (including the overhead of sub-property checking) and its improvement over the method using zChaff. Compared to the method without any learnings (column 2), our spatial decomposition based learning method can drastically reduce the test generation time.

TABLE I. Decomposition Result for MIPS Processor

Property (Tests)	zChaff [10] (sec)	Clus. #	Ref. #	Spatial (sec)	Speedup zChaff vs Spa.
p_1	127.52	3	2	49.41	2.58
p_2	49.24	3	2	15.73	3.13
p_3	9.18	2	1	4.99	1.84
p_4	13.78	2	1	7.28	1.89
p_5	31.63	3	2	12.74	2.48
p_6	120.72	3	2	54.21	2.23

For the remaining 14 properties, we applied both spatial and temporal decomposition individually. Figure 5 illustrates the performance improvement over the method using zChaff. It shows that the temporal method can drastically reduce test generation time (2-4 times). Although spatial decomposition outperforms temporal decomposition in this case study, comparing with the method using zChaff, temporal decomposition can still have significant improvement (2.5 times).

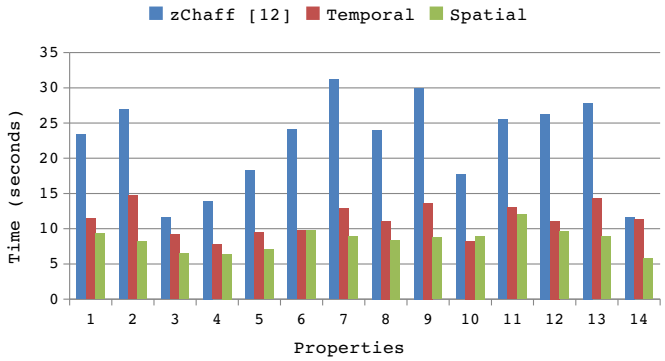


Fig. 5. Test Generation result for MIPS processor

B. A Stock Exchange System

The on-line stock exchange system (OSES) is a software which mainly deals with stock order transactions. The specification of OSES is described by a UML activity diagram which contains 27 activities and 29 transitions. We extract the formal model from the UML specification and transform it to a NuSMV description. We generate 18 complex properties to check various stock transactions. Each transaction is a sequence of activities (events). The test generation for a transaction using only one complex property is time consuming. So we temporally decomposed the transaction into several stages which specify the branch activities along the path, and for each stage we create a sub-property.

Among the 18 complex properties, ten of them are time-consuming (more than 10 seconds without using our method). Table II shows the test generation results for these ten properties using temporal decomposition. The first column indicates the property. The second column indicates the test generation time using zChaff without any decomposition and learning techniques. The third column presents the bound of the complex property. The fourth column indicates the number of temporal sub-properties decomposed along the stock transaction flow. The last two columns indicate the test generation time (using temporal decomposition) and its speedup over zChaff. In this case study, our approach can produce around 3-60 times improvement compared to the method using zChaff.

TABLE II. Decomposition Result for OSES

Prop. (Tests)	zChaff [10] (sec)	Bound	Dec. #	Temporal (sec)	Speedup zChaff vs Temp.
p_1	25.99	8	3	0.78	33.32
p_2	48.99	10	4	2.69	18.21
p_3	39.67	11	5	3.45	11.50
p_4	247.26	11	5	22.46	11.01
p_5	160.73	11	5	15.68	10.25
p_6	97.54	11	4	1.56	62.53
p_7	31.39	10	4	12.31	2.55
p_8	161.74	11	4	12.62	12.82
p_9	142.91	10	4	17.57	8.13
p_{10}	33.77	10	4	1.76	19.19

V. Conclusions

To address the test generation complexity of complex properties using SAT-based BMC, this paper presented a novel method which combines property decomposition and decision ordering based learning techniques. By decomposing a complex property spatially and temporally, we can get a set of sub-properties whose counterexamples can be used to predict the decision ordering for the complex property. Because of the learning from the simple sub-properties, the overall test generation effort can be reduced. The case studies demonstrated the effectiveness of our method using both hardware and software designs that generated significant savings (2-60 times) in overall test generation time.

References

[1] A. Biere, A. Cimatti, and E. M. Clarke. Bounded model checking. *Advances in Computers*, 58, 2003.

[2] H. Koo and P. Mishra. Functional test generation using design and property decomposition techniques. ACM Transactions on Embedded Computing Systems, 8(4), 2009.

[3] M. Chen, X. Qin and P. Mishra. Efficient decision ordering techniques for SAT-based test generation. *DATE*, 490–495, 2010.

[4] H. Lin, J. Jiang, R. Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. *ICCAD*, 32–37, 2008.

[5] M. Chen and P. Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on CAD*, 29(3):396–404, 2010.

[6] J. P. Marques-Silva and K. A. Sakallah. The impact of branching heuristics in propositional satisfiability. *PCAI*, 62–74, 1999.

[7] O. Strichman. Tuning SAT checkers for bounded model checking. CAV, 480–494, 2000.

[8] E. Clarke, O. Grumberg and D. Peled. *Model Checking*. 1999.

[9] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 269–271, 1959.

[10] http://www.princeton.edu/∼chaff/zchaff.html. *zChaff*.

[11] http://nusmv.irst.itc.it/. *NuSMV*.

[12] M. Moskewicz et al. Chaff: Engineering an efficient SAT solver. *DAC*, 530–535, 2001.