View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AIR Universita degli studi di Milano

Decomposition-Based Approach for Model-Based Test Generation

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene

Abstract—Model-based test generation by model checking is a well-known testing technique that, however, suffers from the state explosion problem of model checking and it is, therefore, not always applicable. In this paper, we address this issue by decomposing a system model into suitable subsystem models separately analyzable. Our technique consists in decomposing that portion of a system model that is of interest for a given testing requirement, into a tree of subsystems by exploiting information on model variable dependency. The technique generates tests for the whole system model by merging tests built from those subsystems. We measure and report effectiveness and efficiency of the proposed decomposition-based test generation approach, both in terms of coverage and time.

Index Terms—model-based testing, test case generation, model checking, state explosion problem, decomposition.

1 Introduction

A classical technique for model-based test (MBT) generation exploits the capability of model checkers to produce counterexamples [1], [2]: a *test* is a sequence of states that brings, in the space of reachable states, to one that violates the negation of a *testing goal*.

Due to the well-known "state explosion problem" [3], this approach is not always applicable. Several techniques [4], [5], [6] that have been developed to tackle this problem in the context of property verification, are not suitable for test generation [7] since they may miss parts of the system model that are necessary for building the tests.

In this paper, we present a technique that addresses the state explosion problem for model-based test generation by model decomposition. It works for models given as transition systems. Firstly, given a testing goal, the transition system is decomposed into linked subsystems by exploiting the model variables dependency (one subsystem for each set of interdependent variables). The subsystems constitute a *tree*. Then, a test for the entire system is built by visiting the tree, generating tests for subsystems by suitable testing goals, and merging them. The generation technique has two versions: *StrongTP* assumes that all the tests in all the subsystems have the same length, while its extension, *WeakTP*, allows to generate and merge tests of different length among the subsystems. Both versions are sound, but neither is complete.

The presented approach extends that introduced in [8] in two directions.

1) The test generation technique requires, as input, a dependency tree among subsystems. The original approach in [8] was based on a dependency graph among subsystems, and this caused an extra effort to derive, for every test predicate, a suitable tree

• *The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.*

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

from the graph. Here, the decomposition directly builds a tree dependency structure among subsystems and permits the immediate application of the test generation. Moreover, the current decomposition technique applies not to the entire model but to that portion of it which is of interest for a given testing goal.

2) Although obtained results in terms of time and memory were encouraging, the experimentation in [8] was very limited and the approach was not fully automatic (system decomposition, test predicate generation, and global test construction were hand-made), so preventing a deep and significant comparison analysis. The current version is completely automatic and this made possible a deep evaluation of the proposed technique on a bigger sample of case studies.

Implementation and experimentation of our techniques required to choose a concrete notation for transition systems and a model checker. SCR [9], RSML^{-e} [10], ASMs [11], Statecharts [12], UML behavioural diagrams [13], Event-B [14], SPIN/Promela [15], NuSMV [16], etc., are formal methods suitable to apply our approach.

We applied our approach to 87 NuSMV models, representative of real-life systems. Experiments show that the proposed technique is able to increase the coverage of testing goals by around 3.1 percentage points (*pp*) w.r.t. the classical technique without decomposition; moreover, it speeds up the generation time of around 14%. The technique pays a price in terms of completeness because of the applied decomposition: the percentage of infeasible testing goals increases by around 0.65 *pp* due to the decomposition.

The paper is organized as follows. Sect. 2 provides some basic definitions of transition systems and variable dependency, and it briefly recalls the model-based test generation by model checking. Sect. 3 presents the procedure for system decomposition into a tree of dependent subsystems. Sect. 4 recalls from [8] the strong and weak techniques for test generation. Experimental results about applicability and comparison of the two techniques in test generation are shown in Sect. 5. Sect. 6 identifies possible threats to the validity of the approach, Sect. 7 reviews related literature, and Sect. 8 concludes the paper.

2 Basic Definitions

System models are given in terms of transition systems. Relevant definitions, adapted from [17], are reported in Sect. 2.1. The model-based test generation approach by model checking, and coverage criteria for transition systems are presented in Sect. 2.2.

2.1 Transition System Specifications

Definition 1 (Transition system)**.** A transition system M is a tuple $\langle A, P, \Theta \rangle$ where

- A is a first order structure representing the instantaneous configuration of the system. A has a first order signature G including a finite set of variables $V = \{v_1, \ldots, v_n\}$, a domain D_{v_i} for each variable v_i, relations and functions, and an interpretation function. The system *state* is uniquely determined by the values of the variables.
- P is a program consisting of a sequence of *next assignments* $v'_1 := e_1, \ldots, v'_n := e_n$, being $V' = \{v'_1, \ldots, v'_n\}$ the next state variables; e_i is a term over G and it can contain variables of V and V'.
- $\Theta = \{v_1 = e^0_1, \ldots, v_n = e^0_n\}$ is the set of *initial assignments*; e^0_i can contain only variables of V.

Terms e_i and e^0_i in next and initial assignments may contain conditional expressions. We assume that G may contain a predefined function $random(D)$, randomly returning a value taken from domain D.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering


```
signature A:
V = {SafInject, Overridden, Press, WaterPress, Valve}
D_SafInject = {OFF, OFF_VALVE, ALERT, ON}
D_Overridden = boolean
D_Press = {Low, Mid, High, Unknown}
D_WaterPress = {0, ..., 1000}
D_Valve = {open, closed}

program P:
SafInject' :=
if Press = Low ∧ ¬Overridden ∧ Overridden' then OFF_VALVE
elsif Press = Low ∧ Press' = Normal then ALERT
elsif Press' = High then ON else OFF;
Overridden' := Press = Low ∧ Press' = Unknown;
Press' := if Valve' = open then Unknown
          elsif WaterPress' < 300 then Low
          elsif WaterPress' < 600 then Normal
          else High;
WaterPress' = random(max(0, WaterPress − 5) ...
                     min(1000, WaterPress + 5))
Valve' = random(D_Valve)

initial state Θ:
SafInject = OFF, Overridden = FALSE, Press = Low,
WaterPress = 0, Valve = closed
```

Code 1. Transition system example – SIS

Example 1. As explanatory example, we consider a Safety Injection System (SIS), a simplified version of a control system for safety injection [2]. The SIS is modeled by the transition system $M = \langle A, P, \Theta \rangle$ shown in Code 1. The SIS monitors the *water pressure* (which can change at most of $\pm$ 5 units at each step) and a *valve*. If the valve is open, then the *pressure* level is unknown, otherwise can be low, normal, or high depending on water pressure. The safety system is *overridden* only when the pressure from low becomes unknown. The SIS *injects* coolant when *pressure* is high, it becomes off_valve when it is overridden, it alerts when the pressure becomes normal from low, otherwise it is off.

Definition 2 (Computational step)**.** Executing the program P in a state s consists in evaluating terms $e_1, \ldots, e_n$ in s and assigning the computed values to variables $v_1, \ldots, v_n$ obtaining the next state s'.

Note that, because of variables dependencies, a set of assignments cannot be evaluated in any order. For instance, $x' := y'$ and $y' := x$ can be evaluated only in one order. We suppose that P and Θ are well-defined and thus there always exists an order that permits to evaluate all the assigned terms (there are no *combinatorial loops* [18], i.e., cycles of dependencies not broken by delays). For example, program $P = \{x' := y', y' := x'\}$ is not well-defined as it contains a combinatorial loop among variables $\{x, y\}$.

Definition 3 (System execution)**.** An execution of a transition system is a finite or infinite sequence of states s_0, s_1, ..., s_n such that the initial state s_0 is obtained by evaluating the assignments in Θ and each state s_{i+1} is obtained by executing P at state s_i.

Note that transition systems allow modeling nondeterministic systems. Because of the function $random$, executing P twice from the same state s may lead to two different next states.

Definition 4 (Variable dependency)**.** Given two variables $v_i, v_j \in V$ of a transition system, we say that v_i *directly depends on* v_j if v_j (primed or not primed) occurs in e_i or in e_i^0.

We denote by $DirDep(v)$ the set of variables which v directly depends on.

Definition 5 (Dependency graph)**.** We call *dependency graph* of a transition system M the directed graph $DG = \langle V, E \rangle$, where V is the set of variables of M and $(v, w) \in E$ iff v directly depends on w, i.e., $w \in DirDep(v)$.

Note that the dependency graph can contain cy-

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

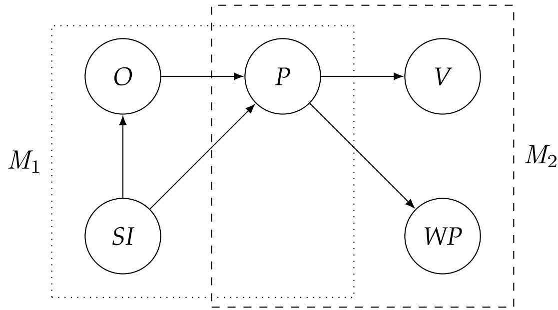


Fig. 1. Variables dependency graph (*O*: *Overridden*, *SI*: *SafInject*, *P*: *Press*, *V*: *Valve*, *WP*: *WaterPress*)

cles, even when a program is well-defined, i.e., it does not contain combinatorial loops. For instance, in a correct program that exchanges two variables x and y by the assignments $x' := y$ and $y' := x$, the two variables are both dependent on the other.

We say that v *depends on* w if there exists a path from v to w in DG. We denote $Dep(v)$ the set of all the variables w, with $w \neq v$, v depends on.

Example 2. Fig. 1 shows the dependency graph of the transition system introduced in Ex. 1. For example, variable $SafInject$ directly depends on $Overridden$ and indirectly depends on $Valve$.

2.2 Model-Based Testing by Model Checking

In model-based testing [19], [20], testing activities exploit a model describing the expected behavior of the system under test.

Definition 6 (Test)**.** A *test* is a finite system execution (as defined in Def. 3).

A test is usually built for covering a given *testing goal*, i.e., a desired system behavior. Testing goals are formally represented by test predicates.

Definition 7 (Test predicate)**.** A *test predicate* is a formula over the model, and determines whether a particular testing goal is reached or not.

Testing goals are usually generated according to some coverage criteria.

Definition 8 (Coverage criterion)**.** A *coverage criterion* C is a function that, given a formal model, produces a set of test predicates. A test suite TS satisfies a coverage criterion C if each test predicate generated with C is satisfied in at least one state of a test sequence in TS.

Some coverage criteria for transition systems are:

- *value coverage*: each value of each variable is covered;
- *decision coverage*: each decision in P and in Θ is covered both to true and to false [21];
- *condition coverage*: i.e, each atomic condition in P and in Θ is covered both to true and to false [21];
- Modified Condition/Decision Coverage (*MCDC*): every atomic condition in a decision (belonging to P or Θ) is shown to independently affect the final value of the decision [22].

Example 3. The value coverage criterion applied to the system shown in Ex. 1 produces the following test predicates: **F**(*SafInject* = *OPEN*), **F**(*SafInject* = *CLOSED*), **F**(*Overridden*), **F**($\neg$*Overridden*), **F**(*Press* = *Low*), ..., **F**(*Press* = *Unknown*), **F**(*WaterPress* = 0), ..., **F**(*WaterPress* = 1000), **F**(*Valve* = *open*), **F**(*Valve* = *closed*).

2.2.1 Test generation by model checking

Model-based test generation by model checking allows automatic generation of test cases from models by exploiting the capability of model checkers to return counterexamples [1], [2]. The technique works as follows. Given a test predicate *tp*, the *trap property* $\neg$*tp* is verified with the model checker, obtaining three possible outcomes:

- The trap property is false, meaning that the test predicate *tp* is *feasible*; the returned counterexample is the test that covers *tp*.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

- The trap property is true, meaning that the test predicate tp is *infeasible* and there is no test that can cover it.
- The model checker terminates without providing any result, usually because of the state explosion problem. In this case, the user does not know whether the test predicate can be covered or not.

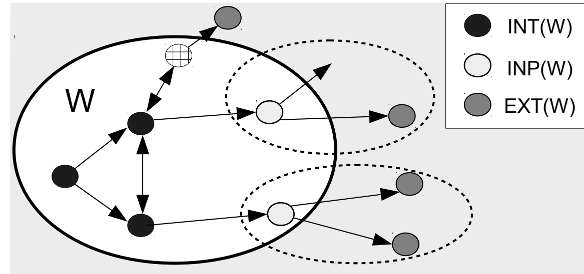


Fig. 2. Variable decomposition

3 System Decomposition

We are interested in decomposing the system M in subsystems in order to improve the test generation process and get a test for the whole system as suitable combination of tests for the single subsystems. The system M must be decomposed in a way that makes the subsystems as much independent as possible from each other and establishes their precise connection links. Since subsystems dependency relies on variables dependency, the dependency graph of M must be analyzed in order to detect those variables that must be kept together – in a unique subsystem – because they are mutually dependent, and those variables that are not part of cyclic dependencies but have dependencies outside the subsystem. The latter represent possible boundary points of the decomposition and their programs can be abstracted.

Before the decomposition process, we place the following definitions identifying classes of variables.

Definition 9. Given a subset $W \subseteq V$ of variables, we call *external* variables of W the variables $EXT(W) = V \backslash W$. Given a variable $v \in W$, we call v as

- *internal* for W if v does not directly depend on any external variable of W, i.e., if $DirDep(v) \cap EXT(W) = \varnothing$; $INT(W)$ identifies the set of internal variables of W;
- *input* for W if it depends only on some external variables of W, i.e., if $\varnothing \subset Dep(v) \subseteq EXT(W)$; $INP(W)$ identifies the set of input variables of W.

Given a variable set W, Fig. 2 depicts the sets $EXT(W)$, $INT(W)$, and $INP(W)$.

Note that a variable v is neither internal nor input if it depends both on some internal variable and on some external variable. The square-patterned variable in Fig. 2 is an example of such kind of variable. Our decomposition process will guarantee to have only internal and input variables.

Starting from M and a test predicate tp, we here present a decomposition process on the subset of V that is of interest for the evaluation of tp – leaving out the rest of the variables – in a way that keeps together the variables in tp, keeps together variables having internal dependencies, and detects those variables (input) that represent possible border points of the decomposition and so linking points between subsets.

Decomposition process

The decomposition process is shown in Alg. 1. Procedure BUILDDECOMP takes in input a test predicate tp over M, and builds a set Dec_{tp} (initially empty) that will contain subsets of V. Then, it adds all variables $var(tp)$ of tp in the initial set V_1 together with their direct dependencies, and calls procedure BUILDSET on V_1 to enlarge the set. The procedure BUILDSET, over a set V_i under evaluation, works as follows:

1) procedure ADDDEPENDENCIES is called on V_i. The procedure checks whether there exists a $v \in V_i$ that is neither internal nor input, and, if any,

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

Algorithm 1 Variables decomposition construction

```
1: procedure BUILDDECOMP(TestPredicate tp)
2:     Dec_tp ← ∅
3:     V_1 ← var(tp) ∪ ⋃_{v∈var(tp)} DirDep(v)
4:     BUILDSET(V_1)
5: procedure BUILDSET(Set V_i)
6:     ADDDEPENDENCIES(V_i)
7:     MERGEINPUTS(V_i)
8:     Dec_tp ← Dec_tp ∪ {V_i}
9:     for all v ∈ INP(V_i) do
10:        BUILDSET({v} ∪ DirDep(v))
11: procedure ADDDEPENDENCIES(Set V_i)
12:    if ∃v ∈ V_i : v ∉ INT(V_i) ∧ v ∉ INP(V_i) then
13:        V_i ← V_i ∪ DirDep(v)
14:        ADDDEPENDENCIES(V_i)
15: procedure MERGEINPUTS(Set V_i)
16:    if ∃v, w ∈ INP(V_i) : Dep(v) ∩ Dep(w) ≠ ∅ then
17:        V_i ← V_i ∪ DirDep(v) ∪ DirDep(w)
18:        MERGEINPUTS(V_i)
```

adds all its direct dependencies $DirDep(v)$ to V_i, and recursively calls itself on the modified set;

2) procedure MERGEINPUTS is called on V_i. The procedure checks whether there are two input variables v and w having common dependencies (i.e., $Dep(v) \cap Dep(w) \neq \varnothing$), and, if any, adds all the direct dependencies of v and w (i.e., $DirDep(v)$ and $DirDep(w)$) to V_i, and recursively calls itself on the modified set;
3) V_i is added to the set Dec_{tp};
4) finally, for each input variable v of V_i, recursively calls itself using, as new set, v and its direct dependencies $DirDep(v)$.

Subsystems construction

Given a transition system $M = \langle A, P, \Theta \rangle$, a test predicate tp over M, and the set Dec_{tp}, we can build a subsystem $M_i = \langle A_i, P_i, \Theta_i \rangle$ of M for each subset V_i in Dec_{tp}, where

- A_i is the structure obtained from A by reducing the set of variables V to V_i;
- P_i contains the next assignments of P for the internal variables $INT(V_i)$, and the next assignment $v' := random(D_v)$ for each input variable $v \in INP(V_i)$;
- Θ_i contains the initial assignments in Θ for the internal variables in $INT(V_i)$, and the initial assignment $v = random(D_v)$ for each input variable $v \in INP(V_i)$.

Each M_i is a well-formed transition system by construction: next and initial assignments in P_i and Θ_i are well-defined and only contain variables of V_i.

Subsystems Dependency Tree

The decomposition in Alg. 1 guarantees that given two subsystems M_i and M_j there is at most one common variable v, input in one subsystem and internal in the other. This leads to a dependency relation among subsystems M_i.

Definition 10 (Subsystems dependency)**.** A subsystem M_i directly depends on another subsystem M_j if $INP(V_i) \cap INT(V_j) = \{v\}$. We call v *linking variable* from M_i to M_j, formally $L(M_i, M_j)$.

The function $DirDep(M_i)$ returns the set of subsystems of M which M_i directly depends on.

The subsystems dependency relation induces a tree structure among subsystems, defined as follows.

Definition 11 (Subsystems Dependency Tree (SDT))**.** The root is given by the subsystem M_1 containing the variables of tp. Each node M_i has as children all subsystems in $DirDep(M_i)$. Leaf nodes are those M_i having $DirDep(M_i) = \varnothing$.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

signature A_1:
$V_1 = \{\mathit{SafInject},\ \mathit{Overridden},\ \mathit{Press}\}$
$D_{\mathit{SafInject}},\ D_{\mathit{Overridden}},\ D_{\mathit{Press}}$ as before in Code 1

program P_1:
$\mathit{SafInject}'$:= as before in Code 1
$\mathit{Overridden}'$:= as before in Code 1
Press' := $\mathit{random}(D_{\mathit{Press}})$

initial state Θ_1:
$\mathit{SafInject}$ = OFF, $\mathit{Overridden}$ = FALSE, Press = Low

Code 2. Transition system example – SIS – Subsystem M_1

signature A_2:
$V_2 = \{\mathit{Press},\ \mathit{WaterPress},\ \mathit{Valve}\}$
$D_{\mathit{Press}},\ D_{\mathit{WaterPress}},\ D_{\mathit{Valve}}$ as before in Code 1

program P_2:
Press' := as before in Code 1
$\mathit{WaterPress}'$ = as before in Code 1

initial state Θ_2:
Press = Low, $\mathit{WaterPress}$ = 0, Valve = closed

Code 3. Transition system example – SIS – Subsystem M_2

Example 4. Let us consider the transition system introduced in Ex. 1 and a test predicate tp such that $var(tp) = \{\mathit{SafInject}\}$. The subsystems obtained through decomposition are $M_1 = \langle A_1, P_1, \Theta_1\rangle$ (shown in Code 2) and $M_2 = \langle A_2, P_2, \Theta_2\rangle$ (shown in Code 3). The linking variable is $L(M_1, M_2) = \mathit{Press}$. Fig. 1 shows the decomposition on the variables dependency graph, and Fig. 3 the corresponding SDT$_{tp}$.

Note that the decomposition technique presented here and based on Alg. 1 improves that presented in [8], since it directly leads to the dependency tree required by the test generation, while the previous one built a dependency graph of subsystems, that had to be adapted to a tree for each test predicate.

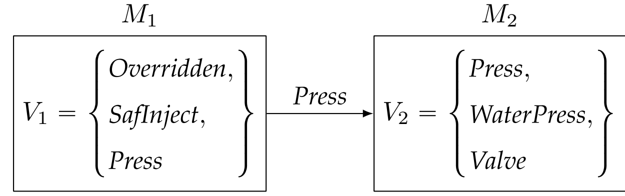


Fig. 3. Transition system example – SIS – SDT$_{tp}$

4 Test Generation by Decomposition

We here recall from [8] the test generation algorithm that computes tests for the whole system by operating on subsystems dependency trees (see Sect. 3).

4.1 Test Generation Algorithm

In order to build a test for covering a test predicate tp (generated by a coverage criterion) for the whole system, the test generation algorithm traverses the dependency tree SDT$_{tp}$ in *pre-order*, builds a partial test for each subsystem, and merges these partial tests together to obtain the final test. Each subsystem M_j builds a test such that the value of the linking variable between M_j and its father subsystem is as requested. The algorithm starts by visiting the root of the tree using tp as test predicate, and it recursively calls itself by visiting the tree nodes with suitable test predicates.

Fig. 4 shows the generation for a generic subsystem K of the tree and a test predicate p:

- It calls the model checker to build a test $\rho = s_0, \ldots, s_n$ to cover the test predicate p.
- If the test is feasible, for each direct dependency M_j of K:
  - It extracts from ρ the input sequence $\mathit{inputSeq}$ for the linking variable $L(K, M_j) = v$ (see Def. 10):

$$\mathit{inputSeq} \leftarrow \pi_v(\rho) = (i_0, \ldots, i_n)$$

    where $\pi_v(\rho)$ yields the projection of ρ with respect to variable v, i.e., $i_k = \pi_v(s_k) = [\![v]\!]_{s_k}$.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

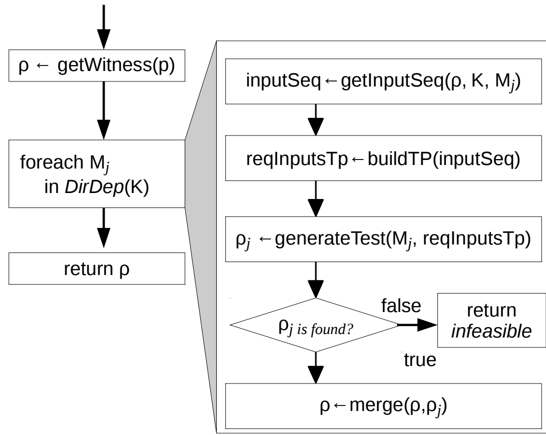


Fig. 4. Test generation approach – generateTest(K, p)

The input sequence represents the inputs provided by M_j to K.

- From *inputSeq*, it computes the test predicate *reqInputsTp* for M_j, defined as LTL property:

$$reqInputsTp \leftarrow in_0 \wedge \mathbf{X}\,(in_1 \wedge \mathbf{X}\,(\ldots \mathbf{X}\,(in_n)\ldots))$$

being $in_j = (v = i_j)$ and $\mathbf{X}$ the *next* temporal connective.

- It recursively visits subsystem M_j, using *reqInputsTp* as test predicate; as a result (if any), it gets the test $\rho_j = s^j_0, \ldots, s^j_n$ for M_j and its dependencies[1].
- If a test ρ_j is returned, it is *merged* with ρ through function `merge` enlarging the states of ρ ($s_h \leftarrow s_h \cup s^j_h$, $h = 0, \ldots, n$); otherwise, it means that the test predicate is considered infeasible.

We call this technique *StrongTP*. Another version of the technique (using a different test predicate structure) is described in the next section.

As an example, Fig. 5 reports a snapshot of the test generation for a system with five subsystems. The algorithm has generated partial tests for M1, M2, and M3, and has already merged the partial tests of M2 and M3; in the recursive visit, the next subsystem that must be visited is M4.

1. Note that ρ_j is guaranteed to be as long as ρ by the test predicate construction.

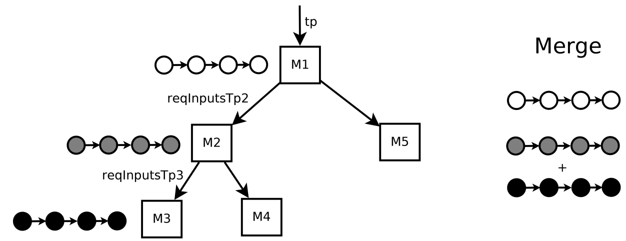


Fig. 5. Test generation example

Example 5. Let us consider the transition system introduced in Ex. 1 and the test predicate $\mathbf{F}(\mathit{SafInject} = \mathit{OFF_VALVE})$. The corresponding decomposition and SDT$_{tp}$ are described in Ex. 4. The test predicate is covered in M_1 by the test

$$\rho_1 = \begin{array}{rcc} \textbf{SafInject}: & \mathit{OFF} & \mathit{OFF_VALVE} \\ \textbf{Overridden}: & \mathit{FALSE} & \mathit{TRUE} \\ \textbf{Press}: & \mathit{Low} & \mathit{Unknown} \end{array}$$

The input sequence is $(\mathit{Low}, \mathit{Unknown})$. The corresponding test predicate for M_2 is:

$$\mathit{Press} = \mathit{Low} \wedge \mathbf{X}\,(\mathit{Press} = \mathit{Unknown})$$

The test predicate is feasible in M_2 and covered by the test

$$\rho_2 = \begin{array}{rcc} \textbf{Press}: & \mathit{Low} & \mathit{Unknown} \\ \textbf{Valve}: & \mathit{closed} & \mathit{open} \\ \textbf{WaterPress}: & 0 & 2 \end{array}$$

The test $\rho = \rho_1 \cup \rho_2$ for the global system is as follows

$$\rho_= \begin{array}{rcc} \textbf{SafInject}: & \mathit{OFF} & \mathit{OFF_VALVE} \\ \textbf{Overridden}: & \mathit{FALSE} & \mathit{TRUE} \\ \textbf{Press}: & \mathit{Low} & \mathit{Unknown} \\ \textbf{Valve}: & \mathit{closed} & \mathit{open} \\ \textbf{WaterPress}: & 0 & 2 \end{array}$$

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

Soundness and Completeness

In [8], we provide the proof that StrongTP is *sound*, i.e., each test produced by the technique is a valid execution (called *allowed sequence* in [8]) of the entire system M. However, in [8], we also show that this technique is *not complete*, i.e., there exists a test predicate not covered by StrongTP that can be covered without decomposition.

In the next section, we provide a different version of the approach that uses a different version of the test predicate *reqInputsTp* and a different way of merging sequences; such modified approach should permit to obtain more completeness.

4.2 WeakTP technique

Technique StrongTP requires that sequences built over the single machines have the same length, i.e., that subsystem K receives, from its children subsystems, the inputs exactly when it requires them. However, the children subsystems may not be able to provide the inputs when requested, but with some delay. We modify technique StrongTP with technique WeakTP, in which children subsystems of K can produce tests ρ_j longer than the test ρ produced over K, and test ρ is *extended* to match the length of tests ρ_j.

In this technique, the test predicate built with function `buildTP` is defined as LTL formula as follows:

$$in_0 \ \mathbf{SXU} \ \ (in_1 \ \mathbf{SXU} \ldots (in_{n-1} \ \mathbf{SXU} \ in_n) \ldots)$$

where $\mathbf{SXU}$ is defined as: $A \ \mathbf{SXU} \ B \equiv A \wedge \mathbf{X}(A \ \mathbf{U} \ B)$, being $\mathbf{U}$ is the *until* temporal connective. $A \ \mathbf{SXU} \ B$ means that A is continuously true for at least one state until B becomes true.

The test ρ_j is at least as long as ρ. ρ_j can be split in $n+1$ sub-sequences $\sigma_0^j, \ldots, \sigma_n^j$ having the same values for the linking variable $L(K, M_j) = v$. Function `merge` merges each state s_t of ρ with all the states of σ_t^j in ρ_j. Note that this can be done only if s_t is *stutter prone* when $|\sigma_t^j| > 1$; a state s is stutter prone if it is a next state of itself, i.e., if, by executing P from s, s can be obtained again.

Example 6. Consider the test predicate $\mathbf{F}(\mathit{SafInject} = \mathit{ALERT})$ for the transition system introduced in Ex. 1. The test predicate is covered in M_1 by the test

$$\rho_1 = \begin{array}{rcc} & \overbrace{}^{s_0^1} & \overbrace{}^{s_1^1} \\ \textbf{SafInject}: & \mathit{OFF} & \mathit{ALERT} \\ \textbf{Overridden}: & \mathit{FALSE} & \mathit{FALSE} \\ \textbf{Press}: & \mathit{Low} & \mathit{Normal} \end{array} \quad (1)$$

The input sequence is $(\mathit{Low}, \mathit{Normal})$. The StrongTP test predicate $\mathit{Press} = \mathit{Low} \wedge \mathbf{X}(\mathit{Press} = \mathit{Normal})$ is infeasible in M_2, since $\mathit{WaterPress}$ cannot reach 300 in one step. Using the WeakTP technique, the corresponding test predicate built for M_2 is

$$\mathit{Press} = \mathit{Low} \ \mathbf{SXU} \ (\mathit{Press} = \mathit{Normal})$$

The test predicate is feasible in M_2 and covered by the test

$$\rho_2 = \begin{array}{rcccc} & \multicolumn{3}{c}{\overbrace{}^{\sigma_0^2}} & \overbrace{}^{\sigma_1^2} \\ \textbf{Press}: & \mathit{Low} & \mathit{Low} & \ldots & \mathit{Normal} \\ \textbf{Valve}: & \mathit{closed} & \mathit{closed} & \ldots & \mathit{closed} \\ \textbf{WaterPress}: & 0 & 5 & \ldots & 300 \end{array}$$

Note that variable Press remains Low in the first 60 states and becomes Normal only in the 61th state. Therefore, we require state s_0^1 of sequence ρ_1 (see Formula 1) to be stutter prone; since this is the case, the technique is applicable.

The test $\rho = \rho_1 \cup \rho_2$ for the complete system is

$$\rho = \begin{array}{rcccc} & \multicolumn{3}{c}{\overbrace{}^{\sigma_0^2 \times s_0^1}} & \overbrace{}^{\sigma_1^2 \times s_1^1} \\ \textbf{SafInject}: & \mathit{OFF} & \mathit{OFF} & \ldots & \mathit{ALERT} \\ \textbf{Overridden}: & \mathit{FALSE} & \mathit{FALSE} & \ldots & \mathit{FALSE} \\ \textbf{Press}: & \mathit{Low} & \mathit{Low} & \ldots & \mathit{Normal} \\ \textbf{Valve}: & \mathit{closed} & \mathit{closed} & \ldots & \mathit{closed} \\ \textbf{WaterPress}: & 0 & 5 & \ldots & 300 \end{array}$$

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

Soundness and Completeness

In [8], we show that WeakTP is sound, but not complete: it covers more test predicates than StrongTP (see Sect. 5.2.2), but still some test predicates not covered by WeakTP can be covered without decomposition.

Example 7. Consider the test predicate $\mathbf{F}(\mathit{SafInject} = \mathit{ON})$ for the transition system introduced in Ex. 1. The test predicate is covered in M_1 by the test

$$\rho_1 = \begin{array}{rcc} & \overbrace{}^{s_0^1} & \overbrace{}^{s_1^1} \\ \textbf{SafInject}: & \mathit{OFF} & \mathit{ON} \\ \textbf{Overridden}: & \mathit{FALSE} & \mathit{FALSE} \\ \textbf{Press}: & \mathit{Low} & \mathit{High} \end{array}$$

The input sequence is $(\mathit{Low}, \mathit{High})$. Both the StrongTP test predicate $\mathit{Press} = \mathit{Low} \wedge \mathbf{X}(\mathit{Press} = \mathit{High})$ and the WeakTP test predicate $\mathit{Press} = \mathit{Low}$ $\mathbf{SXU}(\mathit{Press} = \mathit{High})$ are infeasible in M_2, since Press can not directly go from Low to High.

5 Experiments

In order to evaluate our approach[2], we selected the NuSMV (verification) system [16] for several reasons. First of all, NuSMV models directly reflect the structure and the behavior of transition systems (as defined in Sect. 2.1) in terms of a "possible next state" relation between states that are determined by the values of variables. Moreover, NuSMV comes together with a symbolic model checker (supporting BDD-based model checking of CTL and LTL properties and SAT-based bounded model checking of LTL properties) that can be directly used for test case generation; as a matter of fact, NuSMV is widely used in model-based test case generation [23]. The choice of NuSMV, however, does not limit the validity of our evaluation, as discussed in Sect. 6.

2. Benchmarks and experimental results are available at http://nuseen.sourceforge.net/decompositionBasedTestGenTSE.html.

5.1 Benchmarks

We have gathered 87 NuSMV models from different sources:

- 62 are taken from the NuSMV distribution[3]; note that we do not support 10 out of the 72 models of the NuSMV distribution, as they contain a particular kind of DEFINE alias that does not permit to identify its values statically and so to build the subsystems in the decomposition;
- 20 have been retrieved from different sources on the web (mainly models used in model checking classes);
- 5 have been obtained using the AsmetaSMV tool [24] that translates Abstract State Machine (ASMs) models to NuSMV models; we selected ASM models of real-life case studies as a landing gear system [25], a hemodialysis device [26], and a device for measuring amblyopia [27].

When necessary, models have been `flattened` in order to eliminate modules and parameters. We have used NuSeen[4] [28], a tool framework for NuSMV, for performing different operations: model parsing, dependencies analysis, building of the dependency graph, and the computation of the SDT.

To give an idea of the size and the complexity of the benchmark models, Fig. 6 shows, for each model (a point in the scattered plot), its number of variables and number of states. We observe that, for most of the models, the number of states grows with the number of variables. However, since the state space depends also on the domain size of the variables, the models with the highest number of states are not those with the highest number of variables, and also the other way round.

3. http://nusmv.fbk.eu/

4. http://nuseen.sourceforge.net/

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

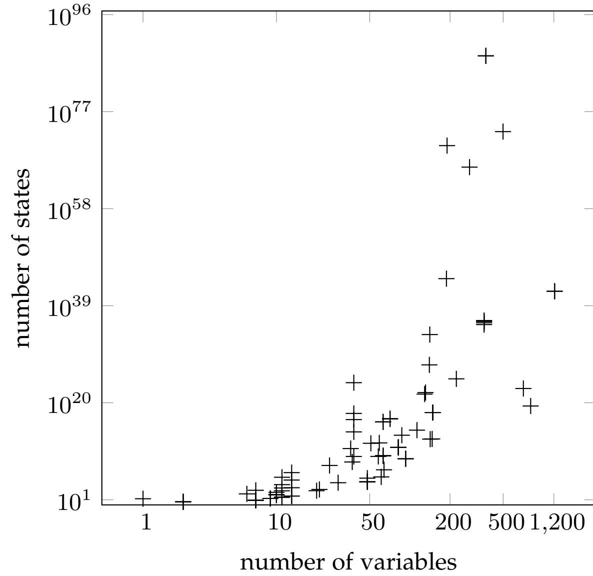


Fig. 6. Benchmarks size

TABLE 1
Number of test predicates

Criterion	Generated	Selected for test generation			
		Total	Max	Min	AVG
DC	6803	6803	348	0	78.20
VC	285072	8849	593	4	101.71
AllTP	291875	15652	831	6	179.91

For all the models, we generated test predicates for achieving decision coverage (DC) and value coverage (VC). The union of all the test predicates is denoted as AllTP. Other coverage criteria, like condition coverage and MCDC, are not supported yet by our implementation and, therefore, they have not been taken into consideration in our analysis; we consider them for future work. Table 1 reports the number of generated test predicates for all the models. As expected, value coverage produces the majority of test predicates, since it builds a test predicate for each value of each variable. In order to keep the number of test predicates tractable for test generation, in value coverage we selected (among those generated) maximum 10 test predicates for each variable (i.e., for variables having more than 10 values, we randomly selected 10 of these values). Table 1 also reports the number of selected test predicates; moreover, for the selected ones, it reports the maximum, minimum, and average number of test predicates over all the models.

5.2 Experimental results

In this section, we evaluate the effectiveness of the proposed test generation approach by assessing how much it is able to mitigate the state explosion problem. We will consider three main measures: the decomposition of the system in Sect. 5.2.1, how much this affects the coverage of test predicates in Sect. 5.2.2, and the generation time in Sect. 5.2.3.

5.2.1 System decomposition

Our approach tries to tackle the state explosion problem by decomposing the system, so that the state space that must be handled by the model checker is smaller. We here compute how much our technique is able to reduce the size of the system under test (in terms of number of states). As size of a decomposed system, we consider the maximum size among its subsystems. Given a decomposed system, we compute the obtained reduction as percentage change $\frac{s_d - s_g}{s_g}$ between the size s_g of the global system and the size s_d of the decomposed system. Fig. 7 shows, for each model, the average size reduction over all the decompositions for all of its test predicates. We observe that, on average, the decomposed system is 46.5% smaller than the original system; we have obtained a maximum reduction of nearly 100% when one subsystem per variable is obtained, and no reduction when the system is not decomposable at all.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

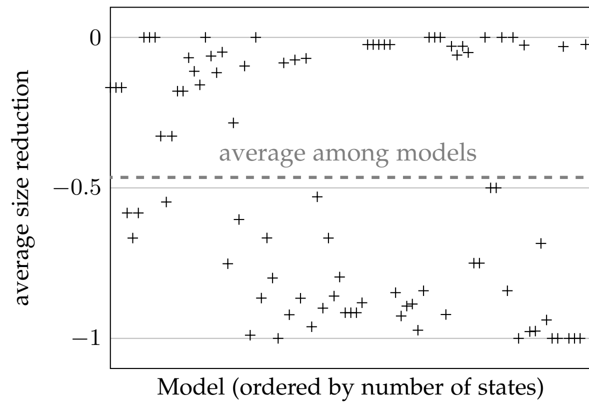


Fig. 7. Size reduction of decomposed systems

5.2.2 Test predicates coverage

We are here interested in measuring if and how much the two techniques based on model decomposition are able to improve the coverage w.r.t. the classical technique without decomposition.

For each test predicate, we generated a test without any decomposition (*NoDecomp*) and by using the *StrongTP* and *WeakTP* techniques. We have applied SAT-based bounded model checking for the $reqInputsTp$ test predicate generated by the StrongTP technique (as we are able to exactly specify the number of steps necessary for violating the property) and BDD-based model checking in all the other cases. For NoDecomp, we have applied the COI abstraction [3] that removes the variables that do not affect the test predicate under test. We fixed a timeout of half an hour for the generation for each test predicate.

Table 2 reports, for the NoDecomp, StrongTP, and WeakTP techniques, the numbers of feasible and infeasible test predicates, and of those for which we were not able to assess anything due to the timeout. The table reports the results for the two coverage criteria and all the test predicates. For StrongTP and WeakTP, the table also reports the percentage change $\frac{A-N}{N}$, being A the result of the decomposition-based technique and N the result of NoDecomp. We can observe for StrongTP and WeakTP a $\approx 4.5\%$ increase of the number of feasible test predicates and a $\approx 17.5\%$ reduction of the number of timeouts; as expected, due to decomposition, the two techniques also increase (by $\approx 6.7\%$) the number of infeasible test predicates (see Sect. 4 regarding the completeness of the approach).

To better evaluate the change (due to the decomposition-based techniques) of the percentages of the feasible, infeasible and timed out test predicates, in Table 3 we report those percentages and, for StrongTP and WeakTP, the *percentage point (pp) change* w.r.t. NoDecomp, computed as the difference between the two percentages A-N, being A the result of the decomposition-based technique and N the result of NoDecomp. StrongTP and WeakTP increase the percentage of covered test predicates by $\approx 3.1pp$ and they reduce the percentage of those that are not covered because of the timeout by $\approx 3.75pp$. Note that they also slightly increase ($\approx 0.65pp$) the percentage of infeasible test predicates; indeed, in addition to those that are found infeasible also in NoDecomp (that are actually impossible to cover), the proposed techniques may fail in generating some tests for some feasible test predicates because of the applied decomposition. It seems that WeakTP performs slightly better than StrongTP: the percentage of feasible test predicates is increased, while the percentages of infeasible and timed out ones are decreased.

We now analyze how the different models contribute to the results. Fig. 8 reports, for each model, the percentage point change in terms of feasible, infeasible, and timed out test predicates of the two decomposition-based techniques w.r.t. NoDecomp (Figs. 8a-8c report the results of StrongTP, and Figs. 8d-8f those of WeakTP). The models are sorted in increasing order by the number of states; for the timeout result (Figs. 8c and 8f), we do not report mod-

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

TABLE 2
Number of Feasible (F), Infeasible (I), and Timeout (TO) test predicates with percentage change w.r.t. NoDecomp (Δ)

	NoDecomp			StrongTP						WeakTP					
	F	I	TO	F		I		TO		F		I		TO	
	#	#	#	#	Δ	#	Δ	#	Δ	#	Δ	#	Δ	#	Δ
DC	6069	535	199	6176	+1.76%	563	+5.23%	64	-67.84%	6175	+1.75%	564	+5.42%	64	-67.84%
VC	4705	988	3156	5050	+7.33%	1064	+7.69%	2735	-13.34%	5117	+8.76%	1059	+7.19%	2673	-15.30%
AllTP	10774	1523	3355	11226	+4.20%	1627	+6.83%	2799	-16.57%	11292	+4.81%	1623	+6.57%	2737	-18.42%

TABLE 3
Percentage of Feasible (F), Infeasible (I), and Timeout (TO) test predicates with percentage point change w.r.t. NoDecomp ($\Delta(pp)$)

	NoDecomp			StrongTP						WeakTP					
	F	I	TO	F		I		TO		F		I		TO	
	%	%	%	%	$\Delta(pp)$	%	$\Delta(pp)$	%	$\Delta(pp)$	%	$\Delta(pp)$	%	$\Delta(pp)$	%	$\Delta(pp)$
DC	89.21%	7.86%	2.93%	90.78%	+1.57	8.28%	+0.41	0.94%	-1.98	90.77%	+1.56	8.29%	+0.43	0.94%	-1.98
VC	53.17%	11.17%	35.67%	57.07%	+3.90	12.02%	+0.86	30.91%	-4.76	57.83%	+4.66	11.97%	+0.80	30.21%	-5.46
AllTP	68.83%	9.73%	21.43%	71.72%	+2.89	10.39%	+0.66	17.88%	-3.55	72.14%	+3.31	10.37%	+0.64	17.49%	-3.95

els without timed out test predicates both in NoDecomp and in the decomposition-based technique. We observe a major increment of feasible test predicates (on average, greater than 10pp) and reduction of timed out ones (on average, lower than -8pp) for biggest models (having between 5×10^{16} and 8×10^{87} states and representing the 30% of the models) for which the state explosion problem is a serious issue. However, it seems that the correlation is negative; indeed, as the size of the model increases, the improvement reduces. Some models are so big that also the decomposed subsystems are not small enough to be handled by the model checker in the given timeout.

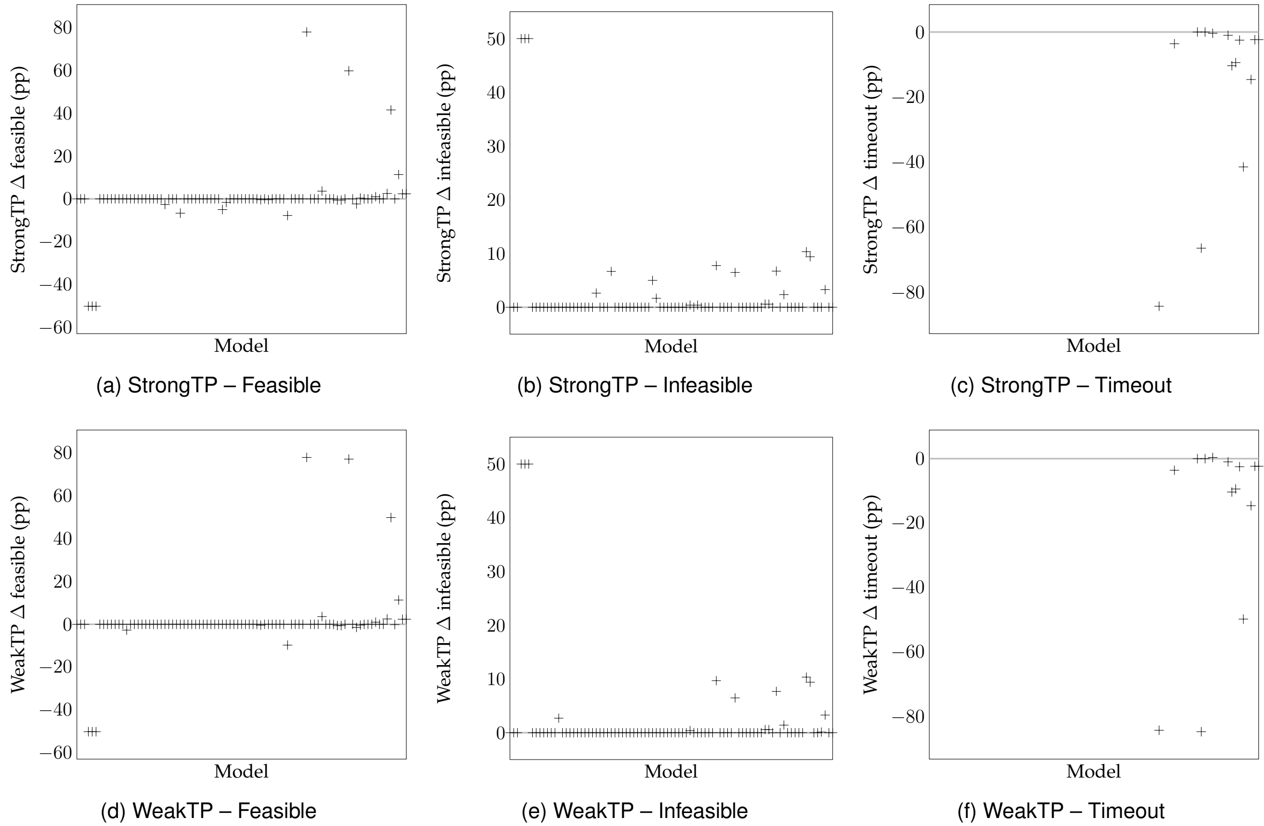
For some models (of different size), the number of feasible test predicates decreases (see Figs. 8a and 8d) and the number of infeasible ones increases (see Figs. 8b and 8e).

The results for the three measures have very similar distributions between StrongTP and WeakTP, except for the infeasible ones (see Figs. 8b and 8e), for which we observe a reduction in some models of average size by WeakTP.

Finally, we are interested in showing how all the test predicates are differently classified (as feasible, infeasible, timeout) between two generation techniques. The results are shown in Fig. 9 by means of Sankey diagrams between NoDecomp and StrongTP, NoDecomp and WeakTP, and StrongTP and WeakTP. In the diagrams, the left side shows how the test predicates are partitioned by the first technique, and the right side shows the partition by the second technique. A line from left to right describes the percentage of all the test predicates that are classified as specified by the starting label in the first technique and as specified by the ending label in the second technique: the width of the line is proportional to the represented value. 3.24% of all the test predicates are not covered by NoDecomp because of the timeout but covered by StrongTP;

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

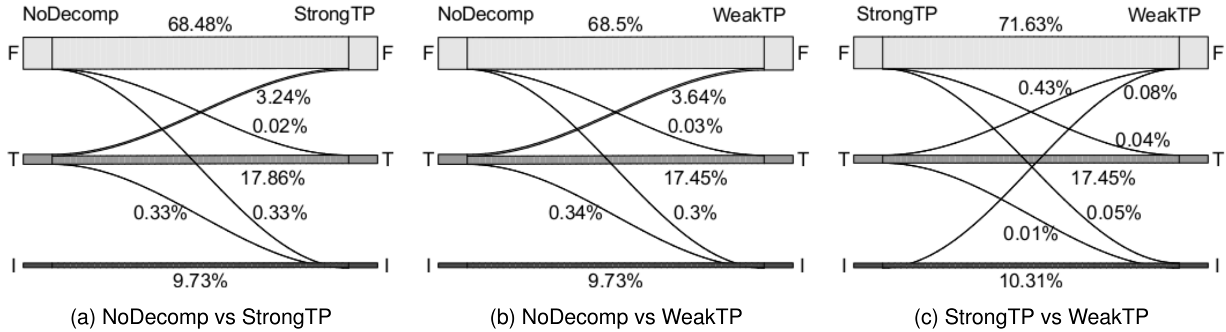
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering



(a) StrongTP – Feasible (b) StrongTP – Infeasible (c) StrongTP – Timeout

(d) WeakTP – Feasible (e) WeakTP – Infeasible (f) WeakTP – Timeout

Fig. 8. Techniques comparison per model (ordered by number of states) – Δ = percentage point change w.r.t. NoDecomp



(a) NoDecomp vs StrongTP (b) NoDecomp vs WeakTP (c) StrongTP vs WeakTP

Fig. 9. Comparison between techniques: baseline vs compared (F: Feasible, T: Timeout, I: Infeasible)

WeakTP improves even more (3.64%). We also observe that the percentage of test predicates that are feasible with NoDecomp and become infeasible because of the decomposition is low: 0.33% for StrongTP and 0.3% for WeakTP. Some test predicates that timed out with NoDecomp are found infeasible (0.33% for StrongTP and 0.34% for WeakTP) by the proposed techniques: these may be either due to the fact that they are actually infeasible or because of the decomposition. There are very rare cases in which feasible test predicates in NoDecomp are timed out with StrongTP (0.02%) and WeakTP (0.03%). We investigated these cases and we

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

TABLE 4
Test generation time (Δ = percentage change w.r.t. NoDecomp)

	NoDecomp	StrongTP		WeakTP	
	h	h	Δ	h	Δ
DC	115.92	62.92	-45.72%	62.92	-45.72%
VC	1852.75	1631.38	-11.95%	1627.10	-12.18%
AllTP	1968.67	1694.31	-13.94%	1690.02	-14.15%

found that these anomalies are due to respectively 3 and 5 test predicates, that they belong to big models with a low size reduction of the decomposed systems, and that the generation time in NoDecomp is very close to the timeout. We therefore cannot exclude that these cases can happen.

WeakTP covers more than StrongTP mainly thanks to test predicates that timed out in StrongTP and become feasible in WeakTP (0.43% of all); the percentage of infeasible ones that become feasible is negligible (0.08% of all).

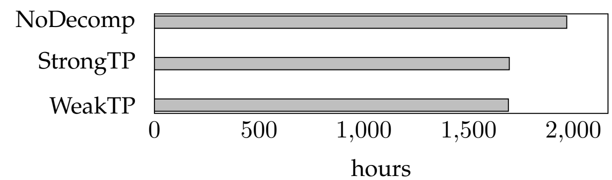
5.2.3 Generation time

The main consequence of the state explosion problem is that the model checking time grows exponentially with the model size. Indeed, symbolic model checking is very time expensive in order to be efficient in representing the state space in a compact way.

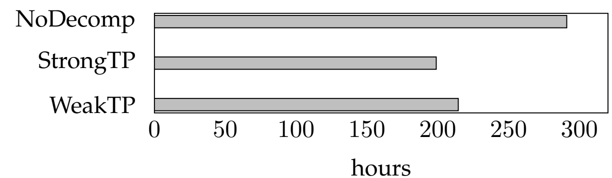
We are here interested in evaluating the test generation time of StrongTP and WeakTP, also in comparison with NoDecomp.

Table 4 reports the time taken (in hours) by the three techniques and, for StrongTP and WeakTP, the percentage change w.r.t. NoDecomp. Results are reported for the two coverage criteria and for all the test predicates (these are also reported in Fig. 10a).

The two proposed techniques improve NoDecomp: StrongTP reduces the time of 13.94% and WeakTP of 14.15%. However, the time reduction depends on the model size; Figs. 11a and 11b show the percentage



(a) All test predicates



(b) No timeout

Fig. 10. Test generation time – Overall results

change of generation time for each model (sorted in increasing order by the number of states): we can see that StrongTP and WeakTP always reduce the generation time for big models, but increase it for small models: this is due to the fact that, when the model is small, the overhead introduced by the techniques is higher than the advantage due to the decomposition.

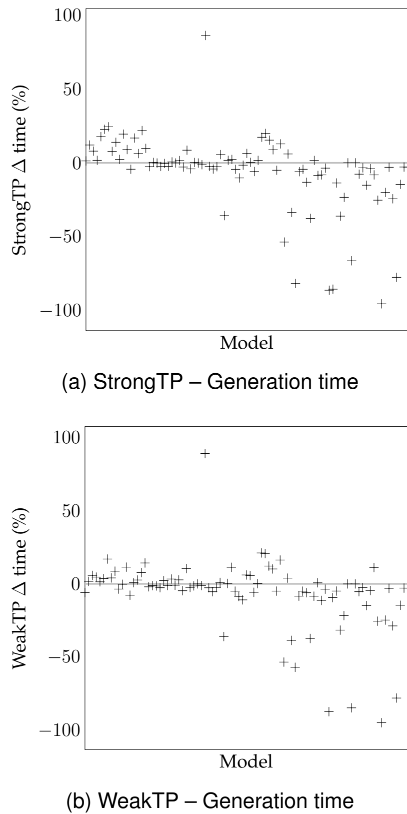
Fig. 10b reports the generation time considering only test predicates not timed out in NoDecomp. It shows that the time saving does not limit to those very complex test predicates that timed out with NoDecomp; indeed, also in this case, StrongTP and WeakTP improve NoDecomp (31.65% by StrongTP and 26.31% by WeakTP).

5.3 Statistical analyses

In this section, we perform some statistical hypothesis testing in order to assess the significance of the results reported in Sect. 5.2. Since we compare measures obtained by two treatments (two generation techniques) applied to the same population (all the test predicates), we are performing a *paired comparison* (also called *crossover*) design [29]. All the statistical hypotheses with corresponding probability value (p-value) and

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering



(a) StrongTP – Generation time

(b) WeakTP – Generation time

Fig. 11. Test generation time per model (sorted by number of states)

result (acceptance, rejection, or not rejection) are reported in Table 5.2.

5.3.1 Test predicates coverage

We first check whether StrongTP and WeakTP significantly modify the number of feasible, infeasible, and timed out test predicates. In order to do this, we apply the non-parametric *McNemar's test* that assesses whether a statistically significant change in proportions have occurred on a dichotomous trait (e.g., "being classified as feasible") between two treatments on the same population [30].

By the results of hypotheses HF1, HF2, HTO1, HTO2, HI1, and HI2, and the results shown in Table 2, we can state that the increase of the number of feasible, the decrease of the number of timed out, and the increase of the number of infeasible test predicates due to the use of both StrongTP and WeakTP w.r.t. NoDecomp are *statistically significant*.

Moreover, by hypotheses HF3 and HTO3, and the results in Table 2, we can state that the increase of feasible and the decrease of timed out test predicates by using WeakTP instead of StrongTP are statistically significant. On the other hand, although in Table 2 we observe that the number of infeasible test predicates slightly decreases in WeakTP, by hypothesis HI3 we cannot assess that WeakTP is significantly more complete than StrongTP.

5.3.2 Generation time

Now, we check whether StrongTP and WeakTP are able to reduce the generation time w.r.t. NoDecomp in a statistically significant way. Since we compare a continuous measure (the time) obtained by two treatments applied to the same population, we can use the parametric hypothesis testing *paired t-test* [29].

By the results of hypotheses HT1, HT2, HT3, and HT4, we can state that both StrongTP and WeakTP are significantly faster than NoDecomp. By the results of hypotheses HT5, HT6, HT7, and HT8, we can state that the same holds also by considering only the test predicates that are feasible with NoDecomp (as done in Fig. 10b). We also compare StrongTP with WeakTP to assess whether one is better than the other in terms of generation time. We cannot reject hypothesis HT9 that the two techniques require the same time. This means that we can elect WeakTP as the best of the two, since it provides a better coverage.

6 THREATS TO VALIDITY

We have identified the following threats to the validity of the empirical evaluation of the proposed approach.

Regarding external validity, it could be that the obtained results can not be generalized to all transition systems. For our experiments, we have used NuSMV,

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

TABLE 5
Statistical testing results

Hypothesis – Name, type (null or alternative), description		result	p-value
Number of feasible, infeasible, timed out test predicates (tps) by McNemar's test			
HF1	Null: StrongTP has on average the same number of feasible tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HF2	Null: WeakTP has on average the same number of feasible tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HF3	Null: WeakTP has on average the same number of feasible tps as StrongTP	rejected	2.025×10^{-11}
HTO1	Null: StrongTP has on average the same number of timed out tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HTO2	Null: WeakTP has on average the same number of timed out tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HTO3	Null: WeakTP has on average the same number of timed out tps as StrongTP	rejected	$< 2.2\times10^{-16}$
HI1	Null: StrongTP has on average the same number of infeasible tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HI2	Null: WeakTP has on average the same number of infeasible tps as NoDecomp	rejected	$< 2.2\times10^{-16}$
HI3	Null: WeakTP has on average the same number of infeasible tps as StrongTP	not rejected	0.5224
Generation time by paired t-test			
HT1	Null: StrongTP has on average the same generation time as NoDecomp	rejected	$< 2.2\times10^{-16}$
HT2	Alternative: StrongTP is faster than NoDecomp	accepted	1
HT3	Null: WeakTP has on average the same generation time as NoDecomp	rejected	$< 2.2\times10^{-16}$
HT4	Alternative: WeakTP is faster than NoDecomp	accepted	1
HT5	Null: StrongTP has on average the same generation time as NoDecomp (considering only the test predicates that are feasible with NoDecomp)	rejected	$< 2.2\times10^{-16}$
HT6	Alternative: StrongTP is faster than NoDecomp (considering only the test predicates that are feasible with NoDecomp)	accepted	1
HT7	Null: WeakTP has on average the same generation time as NoDecomp (considering only the test predicates that are feasible with NoDecomp)	rejected	$< 2.2\times10^{-16}$
HT8	Alternative: WeakTP is faster than NoDecomp (considering only the test predicates that are feasible with NoDecomp)	accepted	1
HT9	Null: WeakTP has on average the same generation time as StrongTP	not rejected	0.1009

but this does not threaten the validity of our results. The mapping from transition systems to NuSMV models is quite straightforward. Different transition system specification languages have been translated to NuSMV, as Statecharts [31], UML behavioural diagrams [13], SCR [2], RSML^{-e} [32], SPIN/Promela [33], and ASMs [24]; five models of our benchmarks have been obtained by translating ASM models in NuSMV models. Therefore, the NuSMV language can be considered as a good representative of different transition system formalisms.

Regarding internal validity [29], we have adopted several precautions with the intent of assuring that the outcomes depend only on the proposed techniques. First, our implementation may not be correct and produce (in a faster way) sequences that are not actual tests of the system. Therefore, we have automatically checked that each generated test is an actual system execution[5]. Moreover, we have performed the experiments on the same computer and using the same model checker settings[6]. We have also

5. We encode each test t as an LTL formula ϕ_t and we use the model checker itself for checking whether ϕ_t describes a system execution.

6. Except for `-bmc` that enables bounded model checking which is part of the StrongTP technique.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

discarded 10 models for which we could not apply our decomposition-based techniques (see Sect. 5.1), so to have exactly the same benchmarks for all the three techniques (i.e., we have a paired comparison design) and be able to apply paired statistical analyses (see Sect. 5.3).

Another threat to internal validity that could impact the results is the chosen timeout of 30 minutes. We have conducted a statistical analysis by varying the timeout from the minimum value (about 24 seconds) that guarantees to cover at least 50% of the test predicates, to maximum 30 minutes. For all the three techniques, we found that the number of test predicates not in timeout grows with the increment of the timeout value. However, the improvements due to StrongTP and WeakTP do not depend on the timeout (the variance of the improvement is around $5{\times}10^{-3}$). Therefore, in our experiments, we report the results with the maximum timeout because this maximizes the number of covered test predicates.

Another threat to the internal validity is due to the threshold on the number of value coverage test predicates, as we may select only the test predicates that are particularly easy/difficult to cover. However, the threshold is applied only to 49 variables out of 11042 total variables; moreover, 94.89% of the non-selected VC test predicates are due to a single model (`nusmvDistributionModels/abp/-abp16_flat.smv`) for which 262100 out of its 262175 value coverage test predicates are discarded. Therefore, we believe that the choice of the threshold does not introduce a bias in the results.

In our experiments, we have always used BDD-based model checking, except for StrongTP for which we applied SAT-based bounded model checking because, in that case, we knew the length of the required counterexamples. By always using SAT-based bounded model checking we could obtain better results in some cases; however, we have already observed that there is no better model checking algorithm for test generation [1]. Moreover, we have used NuSMV as it is integrated in the NuSeen framework that we also use for model decomposition. The new nuXmv implementation may give better results in terms of number of timed out test predicates; however, we believe that it would not affect the drawn conclusions, as it would improve the generation time both for the baseline technique and the proposed decomposition-based techniques.

Regarding construct validity [29], in this paper we do not consider the implementation under test and we cannot evaluate the effectiveness of the generated tests over the implementation (for example, in terms of fault detection or code coverage over the implementation); we refer to [19], [34], [35] for reports on the effectiveness of model-based testing. The test predicates we do not cover because of decomposition could be particularly important for test effectiveness; however, as we cannot evaluate it, we assume that test predicates are equally important and that model coverage is a proxy for test effectiveness. Moreover, we want to point out that the percentage of test predicates that are feasible with NoDecomp and that we do not cover because of the decomposition (0.33% with StrongTP in Fig. 9a, and 0.3% with WeakTP in Fig. 9b) or timeout (0.02% with StrongTP in Fig. 9a, and 0.03 with WeakTP in Fig. 9b) is much lower than the percentage of test predicates that are in timeout with NoDecomp and that we are able to cover (3.24% with StrongTP in Fig. 9a, and 3.64% with WeakTP in Fig. 9b): therefore, StrongTP and WeakTP are performing better than the baseline NoDecomp. This is confirmed by the statistical tests in Sect. 5.3 that assess a statistically significant increment of feasible test predicates using StrongTP and WeakTP.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

7 RELATED WORK

Different approaches have been proposed in the past for handling the state explosion problem in property verification. In the following, we review some of them, compare them with our approach, and discuss whether they could be used also for testing.

The *cone of influence* (COI) technique [3] reduces the size of the model by removing the variables that do not influence the property one wants to check. COI is widely applied: for example, in [36] COI is used for verification of *fFSM* models, a variant of Harel's Statecharts. COI can be useful for test generation when the variables in the test goal have few dependencies; instead, if the test goal involves most of the model variables, COI is not so effective. Actually, our decomposition technique subsumes COI, since the variable decomposition described in Alg. 1 does not consider variables that are not necessary for covering a test predicate.

The *data abstraction* technique [3] creates a mapping between concrete data values and some abstract data values; such mapping is usually able to reduce the state space, but it may not preserve properties. The CEGAR technique [5] is an approach of this kind, that iteratively refines an abstract model. The technique guarantees that, whenever a property is true in the abstract model, it is also true in the initial model; however, if the property is false in the abstract model, the counterexample may represent some behavior in the abstract model not present in the original model (the counterexample is called *spurious* in this case). The spurious counterexample itself is used to refine the abstraction in order to remove the wrong behavior. CEGAR is not suitable for testing because the returned counterexample (that should be used as test) usually does not contain all the variables (due to abstraction), and it may be spurious.

A different abstraction technique has been proposed in [37]. The approach partitions the program into a sequence of subprograms; the *ending state* of a component contains the information to be passed to the next subprogram. The approach performs model checking by visiting the subprograms backwards and trying to prove the property in each subprogram in separation. The decomposition technique differs from ours as we decompose the system in parallel subsystems, while they decompose it sequentially.

Regarding abstraction techniques for test generation, in [38], the approach SMART is presented. SMART performs test generation by decomposing sequential *programs*: given a program, all the functions called in the program are singularly tested, and complete tests are built at the end. The main difference with our approach is that tests for sub-functions are expressed as *summaries* using input preconditions and output postconditions (and not as sequences), and reused when testing higher-level functions. The main advantage is that SMART is both sound and complete compared to the monolithic test generation, while our approaches are only sound. A disadvantage is that SMART must maintain the summaries and it can solve them only at the end. Sometimes constraints on some inputs can not be expressed (for instance a `hash` function) and sometimes all the collected constraints are very hard to solve, leaving some issues still open.

In the past, we already proposed techniques for managing the state explosion problem in model-based test generation by model checking. In [39], we presented a test generation approach for Decomposable by Dependency Asynchronous Parallel (DDAP) systems, which are systems composed of several interleaving subsystems, connected together in a way that the inputs of one subsystem are provided by another subsystem. That approach differs from the current approach on the class of subsystems and on the way

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

to build the test: in [39], the test for the whole system is built by concatenating the tests of the interleaving subsystems, while here the final test is the merge of the tests of the parallel subsystems. We proposed a similar approach in [40], [41] for *sequential net*s of Abstract State Machines (ASMs), which are systems composed of a set of ASMs where only one ASM is active at a time. The approach builds a test suite for every ASM of the net, and then combines these test suites in order to obtain a test suite for the entire system. Also in this case, the main difference w.r.t. the work presented here is that the ASMs in [40], [41] run in sequence, while here the subsystems run in parallel. Moreover, in those works the system was expected to be already decomposed, while here we also provide a decomposition technique.

8 Conclusions

We have proposed a test generation approach, based on model checking, that tries to mitigate the state explosion problem. The approach first decomposes the system under test into a tree of subsystems according to the system variables dependency; the decomposition guarantees that each child in the tree shares a variable with its father: such variable is the input received by the father subsystem from the child subsystem. The test generation consists in visiting the tree in pre-order, generating a test for each subsystem, and merging these tests together in order to obtain a test for the whole system. The approach is sound, although not complete. We proposed two versions of the approach (differing in the test goals that must be covered in the subsystems) that provide different degrees of completeness. Experiments show that the proposed approach is able to increase the coverage of testing goals by around 3 percentage points w.r.t. the classical technique without decomposition in a given timeout of half an hour. Moreover, the approach speeds up the generation time of 14.15%.

Our approach allows the parallelization of the test generation over the single subsystems. As future work, we plan to exploit this opportunity, although this would require an extension of the current technique in the extraction of the input sequences and in the merging of the subsystems tests. Moreover, we plan to evaluate the approach over other coverage criteria as MCDC and condition coverage.

Acknowledgment

We would like to thank the anonymous reviewers for their suggestions and comments that allowed us to greatly improve the paper.

References

[1] G. Fraser and A. Gargantini, “An evaluation of model checkers for specification based test case generation,” in *ICST 2009, 1-4 April 2009, Denver, Colorado, USA*. IEEE Computer Society, 2009, pp. 41–50.

[2] A. Gargantini and C. Heitmeyer, “Using model checking to generate tests from requirements specifications,” in *Proceedings of ESEC/FSE’99*, ser. LNCS, vol. 1687. London, UK: Springer Berlin Heidelberg, 1999, pp. 146–162.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2001.

[4] E. Clarke, W. Klieber, M. Novacek, and P. Zuliani, “Model checking and the state explosion problem,” in *Tools for Practical Software Verification*, ser. LNCS. Springer Berlin Heidelberg, 2012, vol. 7682, pp. 1–30.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, pp. 752–794, 2003.

[6] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005.

[7] W. Prenninger and A. Pretschner, “Abstractions for Model-Based Testing,” *Electron. Notes Theor. Comput. Sci.*, vol. 116, pp. 59–71, Jan. 2005.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

[8] P. Arcaini, A. Gargantini, and E. Riccobene, "Improving model-based test generation by model decomposition," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 119–130.

[9] C. L. Heitmeyer, "Software cost reduction," in *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[10] M. P. E. Heimdahl, S. Rayadurgam, and W. Visser, "Specification Centered Testing," in *Proceedings of the 2nd International Workshop on Automated Program Analysis, Testing and Verification (ICSE 2001)*, 2001.

[11] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[13] L. B. R. dos Santos, E. R. Eras, V. A. de Santiago Júnior, and N. L. Vijaykumar, *A Formal Verification Tool for UML Behavioral Diagrams*. Cham: Springer International Publishing, 2014, pp. 696–711.

[14] J. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[15] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[16] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proceedings International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Springer, July 2002.

[17] D. Peled, *Software Reliability Methods*, ser. Texts in Computer Science. Springer, 2001.

[18] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "NuSMV 2.5 User Manual," http://nusmv.fbk.eu/, 2010.

[19] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.

[20] R. Hierons and J. Derrick, "Editorial: special issue on specification-based testing," *Software Testing, Verification and Reliability*, vol. 10, no. 4, pp. 201–202, 2000.

[21] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. Cambridge University Press, 2008.

[22] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[23] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[24] P. Arcaini, A. Gargantini, and E. Riccobene, "AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications," in *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, ser. LNCS, vol. 5977. Springer, 2010, pp. 61–74.

[25] ——, "Rigorous development process of a safety-critical system: from ASM models to Java code," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 2, pp. 247–269, 2017.

[26] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene, "Integrating formal methods into medical software development: The ASM approach," *Science of Computer Programming*, pp. –, 2017.

[27] ——, "Formal validation and verification of a medical software critical component," in *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, Sept 2015, pp. 80–89.

[28] P. Arcaini, A. Gargantini, and E. Riccobene, "NuSeen: A tool framework for the NuSMV model checker," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 476–483.

[29] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[30] A. Agresti, *An introduction to categorical data analysis, 2nd edition*. NJ: Wiley-Interscience, 2007.

[31] E. M. Clarke and W. Heinle, "Modular Translation of Statecharts to SMV," Carnegie Mellon University, Tech. Rep., 2000.

[32] Y. Choi and M. P. E. Heimdahl, "Model checking RSML-e requirements," in *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering*, ser. HASE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 109–118.

[33] Y. Jiang and Z. Qiu, *S2N: Model Transformation from SPIN to NuSMV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 255–260.

[34] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 392–401.

[35] A. D. Neto, R. Subramanyan, M. Vieira, G. H. Travassos, and F. Shull, "Improving evidence about software technolo-

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2781231, IEEE Transactions on Software Engineering

gies: A look at model-based testing," *IEEE Softw.*, vol. 25, no. 3, pp. 10–13, May 2008.

[36] S. Park and G. Kwon, "Avoidance of state explosion using dependency analysis in model checking control flow model," in *Computational Science and Its Applications - ICCSA 2006: International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part V*, M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganá, Y. Mun, and H. Choo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 905–911.

[37] K. Laster and O. Grumberg, "Modular model checking of software," in *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, B. Steffen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 20–35.

[38] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 47–54.

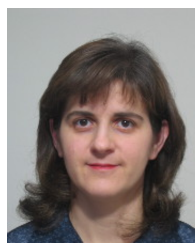
[39] P. Arcaini, A. Gargantini, and E. Riccobene, "An abstraction technique for testing decomposable systems by model checking," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, M. Seidl and N. Tillmann, Eds. Springer International Publishing, 2014, vol. 8570, pp. 36–52.

[40] P. Arcaini, F. Bolis, and A. Gargantini, "Test Generation for Sequential Nets of Abstract State Machines," in *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012), Pisa, Italy, June 18-21, 2012*, ser. Lecture Notes in Computer Science, J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds. Springer, 2012, vol. 7316, pp. 36–50.

[41] P. Arcaini and A. Gargantini, "Test generation for sequential nets of Abstract State Machines with information passing," *Science of Computer Programming*, vol. 94, Part 2, no. 0, pp. 93 – 108, 2014.

Paolo Arcaini is assistant professor at the Charles University, Czech Republic. His research topics include model-based testing, and specification and verification using ASMs.

Angelo Gargantini is associate professor at the University of Bergamo, Italy. His research topics include formal methods, abstract state machines, formal verification, model checking, model-based testing, and combinatorial testing. His home page is http://cs.unibg.it/gargantini/.



Elvinia Riccobene is full professor at the University of Milan, Italy. Her research topics include formal methods, Abstract State Machines, integration between formal modelling and model-driven engineering, model analyses techniques for software systems.

0098-5589 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.