Improving Model-Based Test Generation by Model Decomposition

Paolo Arcaini*
Charles University in Prague
Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz

Angelo Gargantini
Dipartimento di Ingegneria
Università degli Studi di Bergamo, Italy
angelo.gargantini@unibg.it

Elvinia Riccobene
Dipartimento di Informatica
Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

ABSTRACT

One of the well-known techniques for model-based test generation exploits the capability of model checkers to return counterexamples upon property violations. However, this approach is not always optimal in practice due to the required time and memory, or even not feasible due to the state explosion problem of model checking. A way to mitigate these limitations consists in decomposing a system model into suitable subsystem models separately analyzable. In this paper, we show a technique to decompose a system model into subsystems by exploiting the model variables dependency, and then we propose a test generation approach which builds tests for the single subsystems and combines them later in order to obtain tests for the system as a whole. Such approach mitigates the exponential increase of the test generation time and memory consumption, and, compared with the same model-based test generation technique applied to the whole system, shows to be more efficient. We prove that, although not complete, the approach is sound.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

General Terms

Verification

Keywords

Test case generation, model-based testing, state explosion problem, abstraction

*The work was partially supported by Charles University research funds PRVOUK.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786837

1. INTRODUCTION

Model-based test generation by model checking is a well-known technique that allows automatic generation of test cases from models by exploiting the capability of model checkers to return counterexamples [18]. Model checkers are tools that explore the reachable state space of a model and return a counterexample if a property of interest is violated in some state. In the context of model-based testing, once the testing requirements (or *testing goals*) are represented by suitable temporal logic predicates, called *test predicates*, tests are generated by forcing the model checker to build counterexamples upon violation of test predicates negation. Although completely automatic, this technique suffers from the "state explosion problem", i.e., the size of the system state space grows exponentially with the number of variables and the size of their domains, and it could become intractable. Finding a test in an enormous state space is still a challenge.

Much of the research in model checking over the past 30 years has involved developing techniques for dealing with this problem in the context of property verification [14]; however, several of these abstraction techniques (like counterexample guided abstraction [13]) are not suitable for test generation [31]. Indeed, they can guarantee validity of a property in the original model if the property is verified in the abstract model, but they may not guarantee to find the right counterexample if the property is false. Other classical abstractions (like slicing [33] or reduction techniques like *finite focus* [2] that soundly reduces a state machine) transform the original specification to a smaller one for which it may be easier to find the desired tests; however, they may miss parts of the system specification that are necessary for building the tests.

Our goal was to investigate a possible solution in the context of those abstraction techniques for test generation that, following the "divide and conquer" principle, are based on system [4, 5] or property [26] decomposition. Since model checkers suffer exponentially from the size of the system, decomposition brings an exponential gain and allows to test large systems.

In this paper we present a technique for system decomposition and a test generation approach where tests for the whole system are built as combination of tests generated for the subsystems. The idea was inspired from our previous work in [8], although the techniques we propose here for model decomposition and test building are different. A comparison is discussed in Sect. 6.

We here assume that models from which test cases shall be derived are given as transition system formal specifications.

Our approach is based on the following intuitions. Given a system model, a variable dependency graph can be defined on the base of variable updates given by the model assignments. Dependency induces an equivalence relation on the set of model variables and, therefore, variables can be partitioned into equivalence classes representing the strongly connected components of the variables dependency graph. According to the variables decomposition in strongly connected components, the transition system can be decomposed in a set of subsystems that are linked each other by exposing the same dependency relation of the corresponding strongly connected components they are built on. For each transition subsystem, the model-based testing approach by model checking can be used to automatically generate tests, and, at least in principle, it should be more efficient than the same technique applied to the global system. The question is how to exploit the test generation approach applied at subsystem level to build a test for the system as a whole.

For the way most coverage criteria for transition systems are defined, given a test predicate *tp* produced from these criteria, there exists at least a subsystem containing all the variables of *tp*. The test generation for *tp* starts generating a test over this subsystem. This test is then extended by providing suitable test predicates to the other subsystems in order to build (by merging the test generated for the subsystems) a test for the whole system covering the original testing goal.

We here describe how decomposing a transition system into linked subsystems by exploiting the model variables dependency. We then give the algorithm to automatically generate a test covering a given test goal for the global system by merging tests obtained for the subsystems by proving them with suitable testing goals.

We prove that the generation technique is sound, even if not complete. We discuss how to refine the technique to increase its applicability, even if completeness of the refined technique is still difficult to achieve.

Results of the technique application on a certain number of case studies are presented, and these results are compared with those obtained by applying the same technique without system decomposition. Experiments show that a significant benefit is obtained in terms of time and memory.

Note that our approach can be adapted to most state-based methods, as SCR [18], RSML^{-e} [21], ASMs [19], Event-B [1], SPIN/Promela [25], NuSMV [12], etc., since they can be mapped to the formal notation used here.

The paper is organized as follows. Sect. 2 provides some basic definitions of transition systems and briefly recalls the model-based test case generation by model checking. Sect. 3 introduces all necessary concepts and definitions regarding dependency of model variables, variable dependency graph, and system decomposition into dependent subsystems. The algorithm for test generation is presented in Sect. 4, where also soundness and completeness of the technique are discussed. Experimental results about applicability and gain achieved in test generation are presented in Sect. 5. Sect. 6 reviews related literature, and Sect. 7 concludes the paper.

2. BASIC DEFINITIONS

We assume that systems are modeled in terms of transition systems. Therefore, we here provide some basic definitions adapted from [30]. We also briefly recall the approach of automatic test generation from models by model checking, and we introduce coverage criteria suitable for transition systems.

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 $V = \{v_1, v_2, \ldots, v_n\}$, 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 variables in the next state. Each e_i is a term over G, possibly containing variables of V and V'.
- $\Theta = \{v_1 = e_1^0, \ldots, v_n = e_n^0\}$ is the set of *initial assignments*, where e_i^0 can contain only variables of V.

Terms e_i and e_i^0 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.

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 the assignments $x' := y'$ 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 (i.e., there are no *combinatorial loops* [9], that is cycles of dependencies not broken by delays). For example, program $P = \{x' := y', y' := x'\}$ is not well-defined since 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 $s_0, s_1, \ldots, s_n$ 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 the program 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.

Example 1. Consider a locker whose one-digit combination is 4. If the locker *digit* is correct, the locker becomes un*locked*, and then the *handle* can be *OPEN*ed. Once the digit has been set to 4, it cannot be changed. The locker is modeled by the transition system $M = \langle A, P, \Theta \rangle$ shown in Code 1.

Remark 1. Well known state-based formal approaches as SCR [18], RSML^{-e} [21], ASMs [19], Event-B [1], SPIN/-Promela [25], and NuSMV [12] can be represented as transition systems. In some cases, the mapping is straightforward, while other approaches could require suitable conversions.

signature A:
$V = \{handle, locked, digit\}$
$D_{handle} = \{OPEN, CLOSED\}$, $D_{locked} = boolean$,
$D_{digit} = \{0, \ldots, 9\}$

program P:
$handle' :=$ **if** $locked$ **then** $CLOSED$ **else** $random(D_{handle})$
$locked' := digit' \neq 4$
$digit' :=$ **if** $locked$ **then** $random(D_{digit})$ **else** $digit$

initial state Θ:
$handle = CLOSED$
$locked = true$
$digit = 0$

Code 1: Transition system example – Locker

NuSMV specifications, for example, can be easily mapped to transition systems, since in NuSMV the initialization and the update of a variable v have the same form of initial and next assignments of transition systems (i.e., $v = e^0$ and $v' := e$). ASM specifications, instead, require a certain transformation. An ASM model can be viewed (in its simplest form) as a set of transition rules of the form **if** guard **then** Updates **endif** where Updates is a set of updates of locations of the model signature; in order to describe an ASM as a transition system, we should collect, for each location l, all its updates and build a next assignment $l' := e_l$ where e_l is a term containing conditional expressions built from the conditions that guard the updates of l in the ASM model. In SCR, each table can be easily represented as a conditional assignment. SCR events that are used as terms can be translated by using the primed values of variables. For instance, the event @T(x), which means that x becomes true, is equivalent to $x' \wedge \neg x$. Similar transformations can be devised for the other formalisms.

2.2 Model-Based Testing by Model Checking

In model-based testing [24, 32], the specification describing the expected behavior of the system is used for testing purposes.

Definition 4. (**Test**) A *test* is a finite system execution (as defined in Def. 3).

Tests are usually generated for covering some desired system behaviors, called *testing goals*, formally represented by test predicates.

Definition 5. (**Test predicate**) A *test predicate* is a formula over the model, and determines if a particular testing goal is reached.

The generation of testing goals is usually driven by some coverage criteria.

Definition 6. (**Coverage criterion**) A *coverage criterion* C is a function that, given a formal specification, 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.

As *coverage criteria* for transition systems we can identify: *value coverage* (i.e., each value of each variable is covered) and guard coverage criteria [3] as *decision coverage* (i.e., each decision in P and in Θ is covered both to true and to false), *condition coverage* (i.e, each atomic condition in P and in Θ is covered both to true and to false), and Modified Condition/Decision Coverage ($MCDC$) [10], requiring that every atomic condition in a decision (found in P or in Θ) is shown to independently affect the final value of the decision.

Example 2. The value coverage criterion applied to the transition system shown in Ex. 1 produces the following test predicates: $\mathbf{F}(handle = OPEN)$, $\mathbf{F}(handle = CLOSED)$, $\mathbf{F}(locked), \mathbf{F}(\neg locked)$, $\mathbf{F}(digit = 0), \ldots, \mathbf{F}(digit = 9)$.

Test generation by model checking. A classical technique for model-based test generation exploits the capability of model checkers to produce counterexamples [17, 18]. Given a test predicate tp, the *trap property* $\neg tp$ is verified. If the model checker proves that the trap property is false (tp is *feasible*), then the returned counterexample shows how to cover tp. We call the counterexample *witness*, and we translate it to a test. If the model checker explores the whole state space without finding any violation of the trap property, then the test predicate is said *unfeasible* and it is ignored. In the worst case, the model checker terminates without exploring the whole state space and without finding a violation of the trap property (i.e., without producing any counterexample), usually because of the state explosion problem. In this case, the user does not know if either the trap property is true (i.e., the test is unfeasible) or it is false (i.e., there exists a sequence that reaches the goal), and the problem of finding a suitable test for that case remains unsolved.

3. SYSTEM DECOMPOSITION

Variables of a transition system M can be analyzed in order to discover their dependencies and detect the way the system M can be modularized in subsystems. We here first introduce the concepts of variables dependency, dependency graph, and set of strongly connected variables. Then we explain how to decompose a transition system.

Definition 7. (**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 8. (**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., $w \in DirDep(v)$.

We say that v *depends on* w if there exists a path from v to w in DG. The dependency is the transitive closure of the direct dependency. Note that the dependency graph can contain cycles, 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.

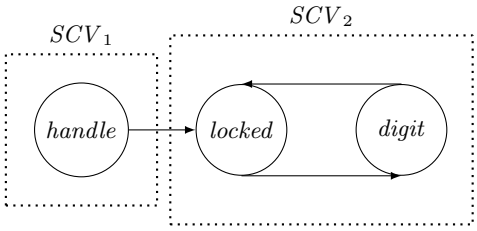


Figure 1: Variables dependency graph

Definition 9. (**Strongly connected variables set**) Given a dependency graph $DG = \langle V, E\rangle$ of a transition system M, each strongly connected component of DG identifies a *strongly connected variables set* (SCV).

Any two variables in one SCV depend one on the other. Intuitively, they constitute a group of interdependent quantities. Furthermore, some variables in an SCV may also directly depend on some variables of other SCVs.

Definition 10. (**SCV inputs**) Given an SCV C, we identify with $IN(C) = \bigcup_{v\in C} DirDep(v)\backslash C$ the *inputs* of C.

$IN(C)$ represents the *inputs* of C since it identifies the direct dependencies (not in C) of the variables of C.

Example 3. Fig. 1 shows the dependency graph and the two SCVs of the transition system introduced in Ex. 1. Variable *handle* directly depends on *locked* and depends on *digit*.

On the base of the decomposition of system variables in strongly connected variables sets, we show how to decompose the transition system.

Decomposition technique. Given a transition system $M = \langle A, P, \Theta\rangle$ and its dependency graph DG, we can build a subsystem $M_i = \langle A_i, P_i, \Theta_i\rangle$ of M for each SCV C_i of DG, where

- A_i is the structure obtained from A by reducing the set of variables V to $V_i = C_i \cup IN(C_i)$;
- P_i contains the next assignments of P for the variables in C_i; moreover, for each variable $v \in IN(C_i)$, P_i contains the next assignment of v in P only if $DirDep(v) = \varnothing$, otherwise it contains $v' := random(D_v)$;
- Θ_i contains the initial assignments in Θ for the variables in C_i; moreover, for each variable $v \in IN(C_i)$, Θ_i contains the initial assignment of v in Θ only if $DirDep(v) = \varnothing$, otherwise it contains $v = random(D_v)$.

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.

We now establish some dependency relations among subsystems. These definitions will be used in the next section for test building.

Definition 11. (**Linking variables**) Given two subsystems M_i and M_j, we call *linking variables* the set of direct dependencies of variables in C_i from variables in C_j, i.e., $L(M_i, M_j) = IN(C_i) \cap C_j$.

Definition 12. (**Subsystems dependency**) A subsystem M_i directly depends on another subsystem M_j if $L(M_i, M_j) \neq \varnothing$. We denote by $DirDep(M_i)$ the set of subsystems of M which M_i directly depends on.

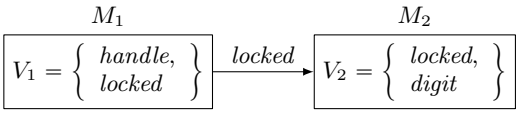


Figure 2: Subsystems dependency graph

The subsystem dependency relation induces an acyclic dependency graph among subsystems.

Example 4. Let us consider the transition system introduced in Ex. 1. The subsystems obtained through decomposition are $M_1 = \langle A_1, P_1, \Theta_1\rangle$:

signature A_1:
$V_1 = \{handle,\ locked\}$

program P_1:
$handle' :=$ **if** *locked* **then** *CLOSED* **else** $random(D_{handle})$
$locked' := random(D_{locked})$

initial state Θ_1:
$handle = CLOSED$
$locked = true$

and $M_2 = \langle A_2, P_2, \Theta_2\rangle$:

signature A_2:
$V_2 = \{locked,\ digit\}$

program P_2:
$locked' := digit' \neq 4$
$digit' :=$ **if** *locked* **then** $random(D_{digit})$ **else** *digit*

initial state Θ_2:
$locked = true$
$digit = 0$

The linking variables are $L(M_1, M_2) = \{locked\}$. The subsystem dependency graph is depicted in Fig. 2.

Remark 2. For the way M_i is built from M, its behavior subsumes the behavior of M restricted to variables V_i. However, M_i may expose further computations that do not correspond to any computation of M, since some of the input variables of M_i in $IN(C_i)$ are randomly initialized or updated and, therefore, they could assume values or sequences of values not allowed in M.

We now provide some definitions to trace back computations of M from computations of its subsystems.

Definition 13. (**State projection**) Given a state s of a transition system and a set of variables $L = \{v_1, \ldots, v_k\}$, we denote by $\pi_L(s)$ the list of values of the variables L in s, i.e., $\pi_L(s) = \{[\![v_1]\!]_s, \ldots, [\![v_k]\!]_s\}$.

Definition 14. (**Sequence projection**) Given a sequence $\rho = s_0, \ldots, s_n$ of a transition system and a set of variables L, the projection of ρ with respect to L is defined as $\pi_L(\rho) = \pi_L(s_0), \ldots, \pi_L(s_n)$.

Definition 15. (**Allowed sequence**) A sequence ρ is *allowed* if there exists an execution ρ' of M such that $\pi_{var(\rho)}(\rho') = \rho$, being $var(\rho)$ the variables occurring in ρ.

Intuitively, a sequence of states is allowed when it is a projection of a valid execution of the entire system M. An allowed sequence ρ may not contain the values for every variable in M, but still, all the variables in ρ correctly behave.

Let ρ be an execution of a subsystem M_i of M. When is ρ an allowed sequence?

THEOREM 1. *If $DirDep(M_i) = \varnothing$, then ρ is allowed.*

PROOF. If a subsystem M_i has no dependencies, its variables set V_i corresponds to C_i (i.e., $IN(C_i) = \varnothing$); therefore, since the initial and next assignments of variables in C_i are the same of M, all the sequences of M_i are allowed. □

THEOREM 2. *If $DirDep(M_i) \neq \varnothing$ and $\pi_{IN(C_i)}(\rho)$ is allowed, then ρ is allowed.*

PROOF. If M_i has some dependencies, then $IN(C_i) \neq \varnothing$. The behavior of the variables in $IN(C_i)$ corresponds to the correct behavior as in M by hypothesis. Since the other variables in C_i are computed as in M, the sequence ρ is a projection with respect to the variables V_i of a valid execution of M. □

Intuitively, an execution of a subsystem M_i is allowed if either M_i has no dependencies, i.e., it is a leaf in the graph, or its inputs represent an allowed behavior, i.e., they force M_i to behave as M.

4. TEST GENERATION BY SYSTEM DECOMPOSITION

We have seen how, exploiting variables decomposition, a transition system can be decomposed in a set of linked subsystems. We show now how to apply the model-based test generation approach by model checking to the single subsystems and how to merge the subsystems tests in order to obtain a test for the global system.

Our technique works on test predicates of a particular class of coverage criteria, called *robust*, defined as follows.

Definition 16. (**Robust criterion**) A coverage criterion C is *robust* to decomposition iff, for each test predicate tp produced by C over M, it exists at least one subsystem M_i of M such that $var(tp) \subseteq V_i$. M_i is said *compatible* with tp.

Most of the classical coverage criteria for transition systems, including all those introduced in Sect. 2.2, are robust. From now on, we assume that the test predicates are derived from robust criteria. In case of *fragile* criteria, techniques for merging subsystems can be applied, but this is left as future work.

Given a test predicate tp that we like to cover for M, we can use as starting point of our test generation technique the subsystem M_i containing all the variables of tp. M_i and its dependencies are sufficient to generate a test able to cover tp (if it exists). If there exists more than one M_i, we choose the subsystem having fewer (direct and indirect) dependencies.

4.1 Test Generation Algorithm

Our test generation algorithm requires that the subgraph consisting of M_i and its dependencies is a tree whose root is M_i. This requirement guarantees that each subsystem provides input values at most to only another subsystem. If the graph were not a tree, a subsystem may be required to provide different values to different subsystems at the same time, so complicating our test generation approach.

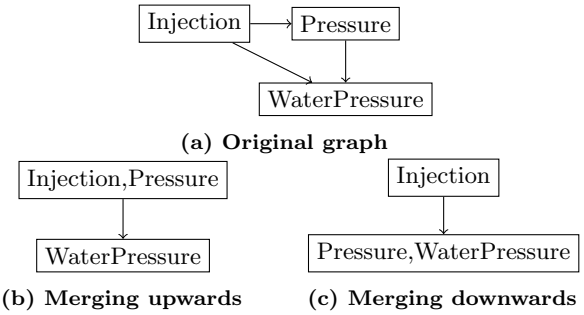


Figure 3: Transforming a graph into a tree with Injection as root

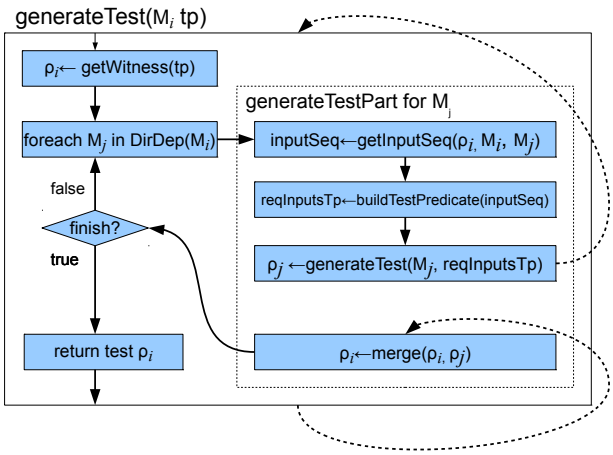


Figure 4: Control flow of the test generation algorithm

If the subgraph does not have a tree-structure, we keep on merging adjacent subsystems (among those reachable from M_i) until M_i becomes the root of a tree. Note that transforming a graph into a tree having M_i as root can be performed in different ways. Fig. 3 shows two possible ways of obtaining the tree: merging the nodes upwards towards the desired root (as in Fig. 3b), or merging the nodes downwards away from the desired root (as in Fig. 3c). In the experiments executed for this paper, we have adopted the latter approach. However, as future work we plan to avoid the transformation to a tree and to generate tests directly from the original dependency graph of subsystems.

The test generation for a test predicate tp is briefly visually represented in Fig. 4. It consists in a recursive function `generateTest` that takes in input a subsystem M_i of M and a test predicate tp which M is compatible with. The function `generateTest` returns as test an allowed sequence for covering tp over the composition of M_i and its (direct and indirect) dependent subsystems. The main steps of the function are the following.

1. It finds ρ_i, a witness for tp in M_i by using a model checker.

Algorithm 1 Test generation algorithm `generateTest` for a subsystem M_i and its test predicates tp

Require: A subsystem M_i
Require: A test predicate tp for M_i
Ensure: A complete test for M_i and its dependencies
1: $\rho_i \leftarrow$ `getWitness`(tp, M_i)
2: **if** $\rho_i =$ `UNFEASIBLE` **then**
3: **return** `UNFEASIBLE`
4: **end if**
5: **for** $M_j \in DirDep(M_i)$ **do**
6: $inputSeq \leftarrow$ `getInputSeq`(ρ_i, M_i, M_j)
7: $reqInputsTp \leftarrow$ `buildTestPredicate`$(inputSeq)$
8: $\rho_j \leftarrow$ `generateTest`$(M_j, reqInputsTp)$
9: **if** $\rho_j =$ `UNFEASIBLE` $\vee$ $\rho_j =$ `UNKNOWN` **then**
10: **return** `UNKNOWN`
11: **end if**
12: $\rho_i \leftarrow$ `merge`(ρ_i, ρ_j)
13: **end for**
14: **return** ρ_i

2. For each dependency M_j, it generates a partial test ρ_j by recursively calling itself. The test ρ_j of the subsystem M_j represents the input sequence ($inputSeq$) to be given to M_i to exhibit the behavior shown by ρ_i.

3. It finally merges the obtained tests with ρ_i in order to find the final test.

The translation of the input sequence ($inputSeq$) to a temporal logic predicate to be used as test predicate is performed by the function `buildTestPredicate`.

The algorithm is precisely described in Alg. 1 and explained in the following. The algorithm traverses the dependency tree in *pre-order*. As first step, the `generateTest` function computes, by means of function `getWitness`, the test $\rho_i = s_0^i, \ldots, s_n^i$ to cover the test predicate tp over M_i (line 1). If the test is not unfeasible, it computes all the direct dependencies of M_i (line 5), and for each M_j of them:

- It extracts from the test ρ_i the input sequence $inputSeq$ for the linking variables $L = L(M_i, M_j) = \{v_1, \ldots, v_k\}$ (see Def. 14) using function `getInputSeq` (line 6):

$$inputSeq = \pi_L(\rho_i) = [\{i_1^0, \ldots, i_k^0\}, \ldots, \{i_1^n, \ldots, i_k^n\}] \quad (1)$$

- Applying function `buildTestPredicate` to the input sequence $inputSeq$ (line 7), it computes the test predicate $reqInputsTp$ for M_j, defined as LTL property as follows:

$$reqInputsTp = in_0 \wedge \mathbf{X}\,(in_1 \wedge \mathbf{X}\,(\ldots \mathbf{X}\,(in_n)\ldots)) \quad (2)$$

being $in_t \equiv \bigwedge_{h=1}^{k} v_h = i_h^t$ $(t = 0, \ldots, n)$, and $\mathbf{X}$ the *next* temporal connective. Note that $reqInputsTp$ is the LTL characterization of the input sequence $inputSeq$. The test predicate has that particular form to obtain a sequence ρ_j (in the next step of the algorithm), such that $\pi_L(\rho_i) = \pi_L(\rho_j)$ with $L = L(M_i, M_j)$.

- It recursively visits subsystem M_j (calling function `generateTest`), using $reqInputsTp$ as test predicate (line 8); as a result (if any), it gets the test $\rho_j = s_0^j, \ldots, s_n^j$ for M_j and its dependencies; note that ρ_j is guaranteed to be as long as ρ_i by the test predicate construction.

- If the returned test ρ_j is neither unfeasible nor unknown, the test ρ_i is *merged* with ρ_j through function `merge`, obtaining the sequence $s_0, \ldots, s_n$ (line 12) where $s_h = s_h^i \cup s_h^j$ $(h = 0, \ldots, n)$, otherwise the test is unknown.

We call this technique *StrongTP*. Another version of the technique will be described in the next section, where the test predicate construction and the merging of the tests are modified.

Example 5. Let us consider the transition system introduced in Ex. 1, whose decomposition is shown in Ex. 4, and the test predicate $\mathbf{F}(handle = OPEN)$. The test predicate is covered in M_1 by the test

$$\rho_1 = \begin{array}{rccc} & \overbrace{}^{s_0^1} & \overbrace{}^{s_1^1} & \overbrace{}^{s_2^1} \\ \textbf{handle}: & CLOSED & CLOSED & OPEN \\ \textbf{locked}: & true & false & false \end{array} \quad (3)$$

The input sequence is $\{locked = true\}$, $\{locked = false\}$, $\{locked = false\}$. The corresponding test predicate for M_2 is:

$$locked \wedge \mathbf{X}\,(\neg locked \wedge \mathbf{X}\,(\neg locked)) \quad (4)$$

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

$$\rho_2 = \begin{array}{rccc} & \overbrace{}^{s_0^2} & \overbrace{}^{s_1^2} & \overbrace{}^{s_2^2} \\ \textbf{locked}: & true & false & false \\ \textbf{digit}: & 0 & 4 & 4 \end{array} \quad (5)$$

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

$$\rho = \begin{array}{rccc} & \overbrace{}^{s_0} & \overbrace{}^{s_1} & \overbrace{}^{s_2} \\ \textbf{handle}: & CLOSED & CLOSED & OPEN \\ \textbf{locked}: & true & false & false \\ \textbf{digit}: & 0 & 4 & 4 \end{array}$$

Soundness and Completeness

A technique is *sound* if each produced test is an allowed sequence (see Def. 15).

THEOREM 3. *The StrongTP technique is* sound.

PROOF. Alg. 1 recursively visits all the (direct and indirect) dependencies of subsystem M_i and builds a test sequence for each subsystem. By Thm. 1, all the sequences produced (at line 1) for the subsystem with no dependencies are allowed: these subsystems are leaves of the dependency graph (the recursive visit stops when a leaf is reached). If the leaves are reached, all the sequences built (at line 1) for their ancestors are allowed by Thm. 2 and the construction of the test predicate at line 7 of the algorithm. The merging of the sequences at line 12 produces an allowed sequence because the common variables between subsystems are guaranteed to be equal. □

To prove *completeness* of the technique, we should prove that each test predicate that can be covered on the global system is also covered by the technique.

THEOREM 4. *The StrongTP technique is not complete.*

PROOF. As counterexample, consider a variation of the Ex. 1 in Code 1 where the next assignment of variable *locked* is modified as $locked' := digit \neq 4$ (i.e., the locker becomes unlocked only *after* the digit is observed to be 4), and still consider $\mathbf{F}(handle = OPEN)$ as test predicate to cover. □

In order to cover some test predicates that are not covered by the StrongTP technique, in the following section we slightly modify the technique, using a different (and less binding) version of the test predicate *reqInputsTp* in Formula 2 and a different way of merging sequences at line 12 of the algorithm.

4.2 WeakTP Technique

Technique StrongTP requires that sequences built over the single machines have the same length (by the test predicate construction) and, therefore, that a given submachine M_i receives, from its dependencies, the inputs it needs *exactly when* it requires them. However, it may be that the dependent subsystems may not be able to provide the required inputs exactly when requested, but *with some delay* (some states later). We modify technique StrongTP with technique WeakTP, in which dependent subsystems of M_i can produce tests ρ_j longer than the test ρ_i produced over M_i, and test ρ_i is *extended* to match the length of tests ρ_j.

In this technique, the test predicate built with function `buildTestPredicate` (from the input sequence in Formula 1) at line 7 of Alg. 1 is defined as LTL formula as follows:

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

being $in_t \equiv \bigwedge_{h=1}^{k} v_h = i_h^t$ $(t = 0, \ldots, n)$, and **SXU** is a new temporal operator defined as follows: $A \ \mathbf{SXU} \ B \equiv A \wedge \mathbf{X}(A \ \mathbf{U} \ B)$ where **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 $\rho_j = s_0^j, \ldots, s_m^j$, computed at line 8 of Alg. 1 with the recursive call of function `generateTest` for covering the test predicate (if feasible), is at least as long as ρ_i (i.e., $m \geqslant n$). ρ_j can be split in $n+1$ sub-sequences $\sigma_0^j, \ldots, \sigma_n^j$ having the same values for the linking variables in $L(M_i, M_j) = \{v_1, \ldots, v_k\}$, i.e.,

$$\rho_j = \overbrace{\underbrace{s_0^j, \ldots, s_{r_1-1}^j}_{in_0}}^{\sigma_0^j}, \overbrace{\underbrace{s_{r_1}^j, \ldots, s_{r_2-1}^j}_{in_1}}^{\sigma_1^j}, \ldots, \overbrace{\underbrace{s_{r_n}^j, \ldots, s_m^j}_{in_n}}^{\sigma_n^j}$$

where, in all the states of each σ_t^j, in_t holds, and $0 < r_1 < r_2 < \ldots < r_n \leqslant m$.

At line 12 of Alg. 1, sequences ρ_i and ρ_j must be merged with function `merge`. Sequences can be merged only if some particular states of ρ_i are *stutter prone*.

Definition 17. Given a transition system $M = \langle A, P, \Theta \rangle$, we call a state s *stutter prone* if, by executing P from s, we can obtain s.

For each subsequence σ_t^j longer than one state (i.e., $|\sigma_t^j| > 1$), state s_t^i of sequence ρ_i must be stutter prone. If this condition is not satisfied, the algorithm returns `UNKNOWN`. Otherwise, sequences ρ_i and ρ_j can be merged as follows:

$$\overbrace{s_0, \ldots, s_{r_1-1}}^{\sigma_0}, \overbrace{s_{r_1}, \ldots, s_{r_2-1}}^{\sigma_1}, \ldots, \overbrace{s_{r_n}, \ldots, s_m}^{\sigma_n} = \overbrace{s_0^j \cup s_0^i, \ldots, s_{r_1-1}^j \cup s_0^i}^{\sigma_0^j \times s_0^i}, \ldots, \overbrace{s_{r_n}^j \cup s_n^i, \ldots, s_m^j \cup s_n^i}^{\sigma_n^j \times s_n^i} \quad (6)$$

where $\sigma_t = \sigma_t^j \times s_t^i$ $(t = 0, \ldots, n)$, i.e., for each s_h of each σ_t, $s_h = s_h^j \cup s_t^i$. Note that we can merge a state s_t^i of ρ_i with all the states of σ_t^j in ρ_j, since s_t^i is stutter prone and, therefore, can be duplicated as many times as necessary.

```
signature A:
V = {handle, locked, digit, cmd}
... D_cmd = {UP, DOWN, NONE}

program P: ...
cmd' := random(D_cmd)
digit' := if cmd = UP then (digit + 1)  mod 10
elseif cmd = DOWN then (digit + 9)  mod 10

initial state Θ: ...
cmd = random(D_cmd)
```

Code 2: Modified running example

Example 6. The proof of Thm. 4 shows a test predicate for the global system that cannot be covered with technique StrongTP (since that technique requires *locked* to become false after one step); the same test predicate, instead, can be covered with technique WeakTP. Using the WeakTP technique, the test predicate built for M_2, starting from the input sequence of the test for M_1 (see Formula 3), is

$$locked \ \mathbf{SXU} \ (\neg locked \ \mathbf{SXU} \ \neg locked)$$

The test predicate is feasible in M_2 and the obtained test is

$$\rho_2 = \begin{array}{rcccc} & \overbrace{}^{\sigma_0^2} & & \overbrace{}^{\sigma_1^2} & \overbrace{}^{\sigma_2^2} \\ \textbf{locked}: & \mathit{true} & \mathit{true} & \mathit{false} & \mathit{false} \\ \textbf{digit}: & 0 & 4 & 4 & 4 \end{array}$$

Note that variable *locked* remains true in the first two states and becomes false only in the third state. Therefore, we require state s_0^1 of sequence ρ_1 (see Formula 3) 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

$$\rho = \begin{array}{rcccc} & \overbrace{}^{\sigma_0^2 \times s_0^1} & & \overbrace{}^{\sigma_1^2 \times s_1^1} & \overbrace{}^{\sigma_2^2 \times s_2^1} \\ \textbf{handle}: & \mathit{CLOSED} & \mathit{CLOSED} & \mathit{CLOSED} & \mathit{OPEN} \\ \textbf{locked}: & \mathit{true} & \mathit{true} & \mathit{false} & \mathit{false} \\ \textbf{digit}: & 0 & 4 & 4 & 4 \end{array}$$

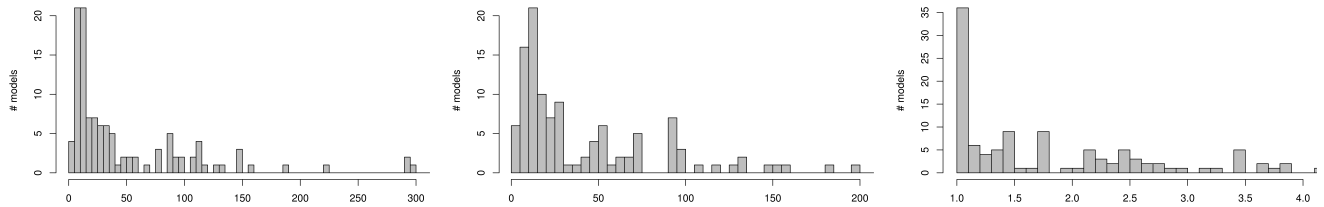
Soundness and Completeness

THEOREM 5. *The WeakTP technique is sound.*

PROOF. If the technique returns a test, it means that in sequence $\rho_i = s_0^i, \ldots, s_n^i$ all the states required to be stutter prone are so. Therefore, ρ_i can be extended to a sequence $\rho_i' = s_0^{i\prime}, \ldots, s_m^{i\prime}$ (by duplicating the states required to be stutter prone) such that, for all the couples $(s_h^{i\prime}, s_h^j)$, the values of linking variables $L(M_i, M_j)$ are the same: this is what we actually do in Formula 6. Then, the proof of Thm. 3 already proves that the composition of two sequences ρ_i' and ρ_j of equal length and with the same values for variables in $L(M_i, M_j)$ is correct. □

THEOREM 6. *The WeakTP technique is not complete.*

PROOF. As counterexample, consider the modification, shown in Code 2, of the transition system described in Ex. 1: a random variable *cmd* over domain $D_{cmd} = \{UP, DOWN, NONE\}$ is introduced to model the transformation of the variable *digit* that can be increased/decreased of only one unit at a time. $\mathbf{F}(handle = OPEN)$ is still the test predicate to cover. The subsystems are as shown in Fig. 5. □



(a) # BBD variables **(b) # variables** **(c) # variables for subsystem (avg)**

Figure 6: Data about the models used in the experiments

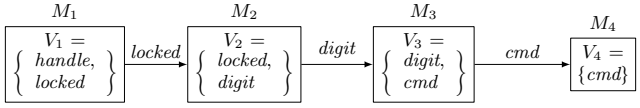


Figure 5: Modified running example – Dependency graph

5. EXPERIMENTS

The proposed technique is in principle more efficient than the model checking approach applied directly to the global system, but it requires several assumptions, like that the system is actually decomposed by using dependency among variables, which may not hold in practice and the actual advantages must be experimentally checked. We present here a series of experiments to validate our approach.

We have performed a set of experiments using a representative example of NuSMV specifications. NuSMV [12] is a well-known tool that performs symbolic model checking. It allows the representation of synchronous and asynchronous finite state systems, and supports model checking of temporal properties. A NuSMV specification describes the behavior of a Finite State Machine (FSM) in terms of a "possible next state" relation between states that are determined by the values of variables. Its language can be easily mapped to the formalism presented in Sect. 2.1. Furthermore, specifications written in ASM, Statecharts, Event-B, SCR, RSML^{-e}, and many other notations have been translated in NuSMV in several previous works [16, 6, 11]. For these reasons, we chose NuSMV for evaluating our approach.

We have gathered 119 NuSMV specifications including examples from the NuSMV site and models we have used in the past for testing a static analysis tool [7]. Then we have `flattened` all the models in order to eliminate modules and parameters. We have reused the parser we have built for NuSeen[1], an eclipse-based framework for NuSMV. To analyze the dependencies, build the dependency graph, and compute the strongly connected variables sets, we have used a feature recently introduced in NuSeen. Fig. 6a shows the sizes of the considered specifications in terms on number of BDD variables (for 115 models, because 4 models having more than 300 BDD variables are not included in the figure): the majority of models have less than 100 BDD variables, but there are still models with more than 100 variables. Our technique, however, works considering the specification variables; therefore, in Fig. 6b we report the distribution of variables in the considered models (for 113 models, because 6 models having more than 200 variables are not included in the figure): the distribution slightly corresponds to the distribution of BDD variables.

We present here some research questions that guided our experiments.

How many systems are decomposable by dependency?

We have applied the decomposition technique presented in Sect. 3 to all the models. Fig. 6c reports the number of variables for subsystem on average over all the models. Most of the models (109/119) have on average less than 4 variables for subsystem. The ideal situation of each subsystem having only one variable occurs for 33 models which originally have, on average, 18.12 variables. Only 5 models were completely not decomposable by our technique. On average, every subsystem has 1/39th of the variables of the entire system. The data shows that dependencies among variables can efficiently guide system decomposition.

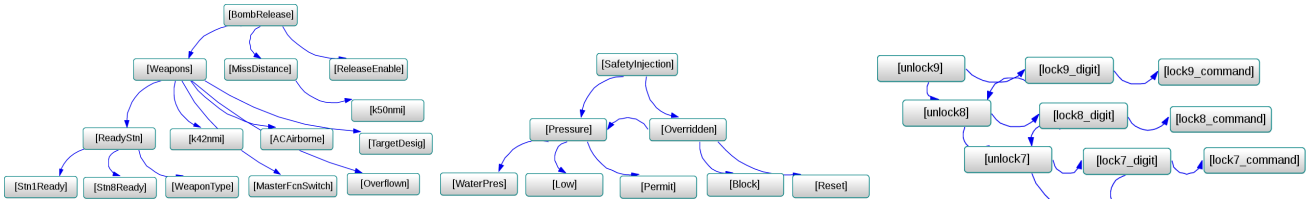
How many dependency subgraphs are trees?

One major assumption of the algorithm presented in Sect. 4.1 is that the subgraph including the subsystem M_i and all its dependencies is a tree. We found that this is true in 52% of all the subsystems we have examined. In all the other cases, the subgraph must be transformed in a tree by merging nodes, as explained in Sect. 4.1. The transformation may increase the complexity of the SCVs and jeopardize the advantages of the decomposition: the worst situation would be when all the subsystems except M_i are merged together, causing a decomposition of the system only in half. We have implemented a simple algorithm that merges SCVs until no more undirected cycles are found and the subgraph becomes a tree. We have then measured the number of vertexes and the average of number of variables in each subsystem. We have found that on average the number of variables for subsystem raises from 4.36 to 7.63, while the number of subsystems decreases from 19 to 11. So, the decomposition is less efficient, but it is still able to reduce the system size by a factor around 10 even for the 48% of the cases in which the dependency subgraph is not already a tree (i.e., from an average of 102 variables in the global system to an average of 7.63 variables per subsystem).

Does test generation actually benefit from system decomposition?

We experimented whether the technique presented in Sect. 4 is really useful for test generation. We took three NuSMV specifications: `BombRel`, the bomb-release component of the flight-control software of an attack aircraft [23], `SIS`, a simplified specification of the control software for the safety injection component of a nuclear power plant [22], and `Lock`

[1] `http://nuseen.sourceforge.net/`



(a) BombRel **(b) SIS** **(c) Lock**

Figure 7: Dependency graphs for the case studies

Table 1: Memory consumption (BDD size) and time (seconds)

Specification	Test predicate	Complete		Decomposition			Δ	
		mem.	time	Technique	mem. (max)	time (sum)	mem.	time
`BombRel` v1	BombRelease = on	363016	5.8	StrongTP	195885	4.04	-46%	-30%
	BombRelease = on & MissDistance ⩾ 10	473027	9.0	WeakTP	333677	4.32	-29%	-52%
`BombRel` v2	BombRelease = on	970498	47.4	StrongTP	646327	39.1	-33%	-17%
	BombRelease = on & MissDistance ⩾ 10	924291	69.1	WeakTP	1058389	40.2	+14%	-41%
`SIS` v1	SafetyInjection = on	547238	6.32	WeakTP	238705	5.68	-56%	-10%
`SIS` v2	SafetyInjection = on	788361	19.5	WeakTP	437094	18.4	-44%	-5%
`Lock` v1	unlock	409813	0.27	WeakTP	1676	0.04	-99%	-86%
`Lock` v2	unlock	547238	6.32	WeakTP	1676	0.08	-98%	-98%

a simple digital lock that requires the insertion of the right sequence of numbers on different keypads in order to unlock [28]. Fig. 7 shows the dependency graphs of the three case studies.

For each specification, we made two versions (v1 and v2) by increasing the size of the variables domains. For instance, in `SIS` the domain of the variable `WaterPressure` is from 0 to 5000 in version v1, and from 0 to 9000 in version v2. For `BombRel`, we choose two different test predicates. The first one simply requires to observe `BombRelease` set to `on`, while the second one also requires that `MissDistance` is at least 10; the second test predicate has been built in a way that technique StrongTP is not able to find a test, but technique WeakTP is.

Table 1 reports the memory required and time took by the classical generation over the complete systems (column *Complete*) and by the proposed generation over the decomposed subsystems (column *Decomposition*). For each considered test predicate, we have used the StrongTP technique when possible, otherwise we have used the WeakTP technique (this has always been applicable since all the states required to be stutter prone were so). The gain differs in the different specifications: this is due both to the considered test predicate and to the degree of decomposition of the system (the more the variables are distributed among the subsystems, the better it is). Using the proposed technique greatly diminishes the required memory and time (column Δ) in all the cases, except in one case in which the use of memory is increased probably due to the higher complexity of the test predicate containing the **SXU** operator.

Note that our technique might not be able to cover a test predicate that is feasible: this may weaken the fault detection with respect to the model checking approach applied directly to the global system (that, however, might also not be able to cover the test predicate for the state explosion problem). Nonetheless, whenever our technique is complete, the fault detection and the coverage provided by the test cases are the same as those obtained by using the model checking approach on the global system.

6. RELATED WORK

Since our approach is based on model checking, one may immediately think of reusing abstraction techniques introduced for formal verification. For this reason, we initially compare our work to the research done in the area of abstractions for property verification.

The *cone of influence* (COI) technique [15] reduces the size of the transition graph by removing from the model the variables that do not influence the variables in the property one wants to check. In [29], COI is used to reduce the state space of *fFSM* models, a variant of Harel's Statecharts; models that could not be verified before, have been verified successfully after its application. COI works well also for test generation, but only if the variables in the property to be verified have few dependencies. For subsystems deep inside the dependency graph, COI is unable to reduce the specification. Actually, our technique subsumes COI, since we also remove variables that are not necessary for covering a test predicate.

The *data abstraction* technique [15], instead, consists in creating a mapping between the data values and a small set of abstract data values; the mapping, extended to states and transitions, usually reduces the state space, but it may not preserve properties. In [13], a technique called CEGAR is presented, to iteratively refine an abstract model. The technique assures that, if a property is true in the abstract model, so it is in the initial model; if it is false in the abstract

model, instead, the *spurious* counterexample may be the result of some behavior in the abstract model not present in the original model. The counterexample itself is used to refine the abstraction so that the *wrong* behavior is eliminated. CEGAR is not suitable for testing: indeed, the returned counterexample usually does not contain all the variables since the abstraction *removes* specification parts, and it may be spurious.

A technique for sequential *modular* decomposition for property verification of complex programs is presented in [27]. The approach consists in partitioning the program into sequentially composed subprograms (instead of the typical solution of partitioning the design into units running in parallel). Based on this partition, the authors present a model checking algorithm for software that arrives at its conclusion by examining each subprogram in separation. They identify *ending states* in the component where the computation is continued in another component and some information passed to the next subprogram. The algorithm then tries to formally prove the property in each component finding the necessary assumptions about the initial (entering) states of the component. The algorithm proceeds backwards until it finds that the property is true in every sub-component starting from any initial state of the system. Since the goal is formal verification, the algorithm must check that the property holds in *any* state, while in our approach disproving a property is not enough since we want to find a counterexample, i.e., a path leading to interesting states in which a suitable property is false. Moreover, we decompose the entire system in subsystems that run in parallel and not sequentially.

There exist few abstraction techniques that are suitable for test generation. Reduction techniques like *finite focus* [2] soundly reduce the original specification to a smaller one for which it may be easier to find the desired tests. Finite focus maps variables with large or unbounded domains to a fixed subset of possible values. In this case the number of variables to be considered is not reduced but their domains are. In order to avoid unsound counterexamples, some constraints must be added and there is not guarantee that the specification is actually simpler than the original one. Moreover, how to reduce domain sets may be a difficult task and no algorithm is given for that. Finite focus could be used also in conjunction with our approach when generating the tests for a single subsystem. In general, our technique is compatible with all the reduction abstractions.

An approach performing test generation by decomposing sequential *programs*, called SMART, is presented in [20]. It proposes a sequential decomposition technique: given a program calling several functions inside it, these called functions are tested in isolation and complete tests are composed only at the end. The main difference with our approach is that tests for sub-functions are not real tests but they are expressed as *summaries* using input preconditions and output postconditions, and then re-used when testing higher-level functions. The main advantage is that SMART is both sound and complete compared to 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.

The approach we presented in [8] shares with this work the idea of exploiting system decomposition to tackle the limitation of model-based test generation by model checking. It presents a technique to build test for Decomposable by Dependency Asynchronous Parallel (DDAP) systems, which are systems composed of several subsystems running in parallel and connected together in a way that the inputs of one subsystem are provided by another subsystem. Apart sharing the philosophical idea of tacking a problem by decomposing it into smaller problems, the approaches differ (1) on the way a system is decomposed, (2) on the class of obtained dependent subsystems (that in [8] are interleaving subsystems, while here we have parallel subsystems), and (3) on the way a test for the whole system is built on the base of the tests for the subsystems (concatenation of tests in [8], merging of tests here).

In [4], we proposed a test generation technique for *sequential net*s of Abstract State Machines (ASMs), which represent systems constituted by a set of ASMs such that only one ASM is active at a time. Given a net of ASMs, a test suite for every ASM in the net is built, and then the tests are combined in order to obtain a test suite for the entire system. The technique has been later extended in [5] for handling the passing of information between subsystems. Apart the different notation, the main difference with this work is that in [4, 5] we suppose to already have the decomposed subsystems, whereas here we propose a way to decompose the global system. Moreover, in [4, 5] the subsystems run in sequence, while here they run in parallel.

7. CONCLUSIONS

We have proposed a test generation approach by model checking that decomposes systems into dependent subsystems on the base of the system variables dependency. Such dependent subsystems coming from the system decomposition can be viewed as systems linked each other – on the base of the variables dependency – in a way that (part of) the inputs of one subsystem are provided by another subsystem. Therefore, a test of the global system can be built by suitably merging tests of the single subsystems. Such approach permits to mitigate the state explosion problem of model checking since the time and the memory required to build a test for each subsystem is considerably less than the time and the memory required when considering the system globally. The method has been proved to be sound but not complete, and its efficiency w.r.t. the same model-based testing approach without system decomposition has been shown by a number of experiments on NuSMV models.

Currently, we automatically build the dependency graphs, but not the subsystems and the test predicates. As future work, we plan to completely automatize our approach and to consider test predicates possibly involving more than one subsystem. This will allow us to perform a larger evaluation on more models across a variety of sizes. Moreover, we plan to better investigate which is the best solution for merging the subsystems when their dependency graph is not a tree.

8. REFERENCES

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

[2] P. Ammann and P. Black. Abstracting formal specifications to generate software tests via model

checking. In *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, volume 2, pages 10.A.6–1–10.A.6–10 vol.2, 1999.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

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

[5] P. Arcaini and A. Gargantini. Test generation for sequential nets of Abstract State Machines with information passing. *Science of Computer Programming*, 94, Part 2(0):93 – 108, 2014.

[6] 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)*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.

[7] P. Arcaini, A. Gargantini, and E. Riccobene. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering*, 7(2):97–107, 2011.

[8] P. Arcaini, A. Gargantini, and E. Riccobene. An abstraction technique for testing decomposable systems by model checking. In M. Seidl and N. Tillmann, editors, *Tests and Proofs*, volume 8570 of *Lecture Notes in Computer Science*, pages 36–52. Springer International Publishing, 2014.

[9] 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.

[10] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[11] Y. Choi and M. P. E. Heimdahl. Model checking RSML-e requirements. In *7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002), 23-25 October 2002, Tokyo, Japan*, pages 109–118. IEEE Computer Society, 2002.

[12] 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)*, volume 2404 of *Lecture Notes in Computer Science*. Springer, July 2002.

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

[14] E. Clarke, W. Klieber, M. Novacek, and P. Zuliani. Model checking and the state explosion problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[16] E. M. Clarke and W. Heinle. Modular Translation of Statecharts to SMV. Technical report, Carnegie Mellon University, 2000.

[17] 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*, pages 41–50. IEEE Computer Society, 2009.

[18] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162, London, UK, 1999. Springer Berlin Heidelberg.

[19] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.

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

[21] M. P. E. Heimdahl, S. Rayadurgam, and W. Visser. Specification Centered Testing. In *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification (ICSE 2001)*, 2001.

[22] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[23] K. L. Heninger, J. Kallander, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington DC, Nov. 1978.

[24] R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.

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

[26] H.-M. Koo and P. Mishra. Functional test generation using design and property decomposition techniques. *ACM Trans. Embed. Comput. Syst.*, 8(4):32:1–32:33, July 2009.

[27] K. Laster and O. Grumberg. Modular model checking of software. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 1998.

[28] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153, Princeton, 1956.

[29] S. Park and G. Kwon. Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model. In *ICCSA 2006*, volume 3984 of *Lecture Notes in Computer Science*, pages 905–911. Springer, 2006.

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

[31] W. Prenninger and A. Pretschner. Abstractions for Model-Based Testing. *Electron. Notes Theor. Comput. Sci.*, 116:59–71, Jan. 2005.

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

[33] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*,