An Abstraction Technique for Testing Decomposable Systems by Model Checking

Paolo Arcaini[1], Angelo Gargantini[1], and Elvinia Riccobene[2]

[1] Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
{paolo.arcaini,angelo.gargantini}@unibg.it
[2] Dipartimento di Informatica, Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. Test generation by model checking exploits the capability of model checkers to return counterexamples upon property violations. The approach suffers from the *state explosion problem* of model checking. For property verification, different abstraction techniques have been proposed to tackle this problem. However, such techniques are not always suitable for test generation. In this paper we focus on Decomposable by Dependency Asynchronous Parallel (DDAP) systems, composed by several subsystems running in parallel and connected together in a way that the inputs of one subsystem are provided by another subsystem. We propose a test generation approach for DDAP systems based on a decompositional abstraction that considers one subsystem at a time. It builds tests for the single subsystems and combines them later in order to obtain a global system test. Such approach avoids the exponential increase of the test generation time and memory consumption. The approach is proved to be sound, but not complete.

1 Introduction

Test generation by model checking is a well-known technique that exploits the capability of model checkers to efficiently explore the state space and build a counterexample when a property is falsified by the model. One main problem is the "state explosion problem", i.e., the size of the system state space grows exponentially w.r.t. the number of variables and the size of their domains. 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 [8]. There exist several abstraction techniques (like counterexample guided abstraction [7]) that address this problem for property verification, but they are not suitable for test generation [19]. 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 [21] or reduction techniques like *finite focus* [1] that soundly reduces a state machine) reduce 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.

The approach presented here can be viewed in the context of those abstraction techniques for test generation that, following the "divide and conquer" principle, are based on system [2,3] or property [16] 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 focus on systems that can be decomposed in two (or more) subsystems that run asynchronously in parallel but such that (part of) the inputs of one subsystem are provided by another subsystem. For such systems, we propose a test generation approach based on model checking, exploiting the decomposition by dependency abstraction. The approach consists in generating the tests for the single subsystems and combining them later, in order to build a test for the whole system. The generation is performed by considering the dependency relation, starting from the "most" dependent subsystem, to the independent subsystem. Such approach permits to exponentially reduce the test generation time and the memory consumption with respect to the basic approach that builds a test for the whole system.

Section 2 provides some background on Kripke structures with inputs, on their representation in the model checker NuSMV. Test case generation by model checking is also briefly recalled in this section. Section 3 introduces DDAP systems, i.e., systems having two components in a dependency relation, and Section 4 proposes a test generation approach for them. Section 5 extends the approach to n-DDAP systems, i.e., systems having more than two components. Preliminary experiments are presented in Section 6. Section 7 reviews some related literature, and Section 8 concludes the paper.

2 Background

We here report some basic concepts regarding the formal structure and the test generation approach by model checking that represent the fundamentals of the theory of DDAP systems, developed in Sections 3, 4, and 5.

2.1 Kripke structures

In this paper we use Kripke structures with inputs [15], that can be conveniently used to represent reactive systems.

Definition 1 (Kripke structure with inputs). *A* Kripke structure with inputs *is a 6-tuple $M = \langle S, S^0, IN, OUT, T, \mathcal{L} \rangle$ where*
- *S is a set of states;*
- *$(S^0 \subseteq S) \neq \emptyset$ is the set of initial states;*
- *IN and OUT are disjoint sets of atomic propositions;*
- *$T \subseteq S \times \mathcal{P}(IN) \times S$ is the transition relation; given a state s and the applied inputs I, the structure moves to a state s', such that $(s, I, s') \in T$.*
- *$\mathcal{L} : S \to \mathcal{P}(OUT)$ is the proposition labeling function.*

Definition 2 (Input sequence). *An* input sequence *for a Kripke structure with inputs is a (possibly infinite) sequence of inputs* $I_0, \ldots, I_n, \ldots$ *with* $I_i \in \mathcal{P}(IN)$.

Definition 3 (Trace). *Given an input sequence* $I_0, \ldots, I_n, \ldots$, *a* trace *for a Kripke structure with inputs is a sequence* $s_0, I_0, s_1, \ldots, s_n, I_n, s_{n+1} \ldots$ *such that* $s_0 \in S^0$ *and* $(s_i, I_i, s_{i+1}) \in T$.

Definition 4 (Test). *A* test *for a Kripke structure with inputs is a finite trace* $s_0, I_0, s_1, \ldots, s_{n-1}, I_{n-1}, s_n$.

We define the set of atomic propositions as $AP = IN \cup OUT$ and CTL/LTL formulae are defined over AP.

Kripke structures with inputs differ from classical Kripke structures because the inputs are explicitly not part of the state and cannot be modified by the machine. However, since for every Kripke structure K with inputs there is a corresponding Kripke structure K' without inputs [5], all the model checking techniques can be still applied.

2.2 Encoding Kripke structures with inputs in NuSMV

NuSMV [6] is a well-known tool that performs symbolic model checking. It allows the representation of synchronous and asynchronous finite state systems, and the analysis of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). 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. A variable type can be Boolean, integer defined over intervals or sets, or an enumeration of symbolic constants. A *state* of the model is an assignment of values to variables.

There are two kinds of variables: *state* variables, declared in the section **VAR**, and *input* variables, declared in the section **IVAR**. The value of state variables can be determined in the **ASSIGN** section in the following way:

ASSIGN var := simple_expression −− *simple assignment*
ASSIGN **init**(var) := simple_expression −− *init value*
ASSIGN **next**(var) := next_expression −− *next value*

A simple assignment determines the value of variable *var* in the current state, the instruction **init** permits to determine the initial value(s) of the variable, and the instruction **next** is used to determine the variable value(s) in the next state(s).

Input variables represent inputs of the system, and their value cannot be bound as done for state variables. They can only be used to determine the next value of state variables.

A **DEFINE** statement (**DEFINE** id := exp) can be used as a macro to syntactically replace an identifier *id* with the expression *expr*.

NuSMV offers another more declarative way of defining initial states and transition relations. Initial states can be defined by the keyword **INIT** followed by characteristic properties that must be satisfied by the variables values in the initial states. Transition relations can be expressed by constraints, through the keyword **TRANS**, on a set of *current state/next state* pairs.

Temporal properties are specified in the **LTLSPEC** (resp. **CTLSPEC**) section that contains the LTL (resp. CTL) properties to be verified.

NuSMV can be used to describe Kripke structures with inputs. The inputs are modeled as input variables (**IVAR**), and the outputs as state variables (**VAR**) or definitions (**DEFINE**).

In this paper we use NuSMV, but the approach is general and applicable to any model checker.

2.3 Model-based test generation by model checking

In model based testing [14,20], the specification describing the expected behavior of the system is used to generate tests that exhibit some desired system behaviors (*testing goals*). Test goals can be 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.*

A classical technique for model-based test generation exploits the capability of model checkers to produce counterexamples [10,12]. If a test predicate can be expressed as a CTL/LTL formula over the model states, then a test suite *covering* the test goals corresponding to a desired coverage criterion can be generated as follows.

1. The test predicates set $\{tp_1, \dots, tp_n\}$ is derived from the specification according to the desired testing goals. Test predicate structure depends on the particular desired coverage criteria [11].
2. For each test predicate tp_i, the *trap property* $\neg tp_i$ is verified. If the model checker proves that the trap property is false (tp_i is *feasible*), then the returned counterexample shows how to cover tp_i. 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).

Note that the specification is used also to produce a test oracle to assess the correctness of the implementation.

Example 1. Consider as example a simple system in which there is a statement like **if** C **then** .... A possible test goal, requiring that the condition is covered by at least a test, can be formalized by the LTL test predicate **F**(C), requiring that C is eventually true. If the model checker finds a counterexample for the trap property !**F**(C), such counterexample leads the system to a state where C is true and, therefore, it is the desired test.

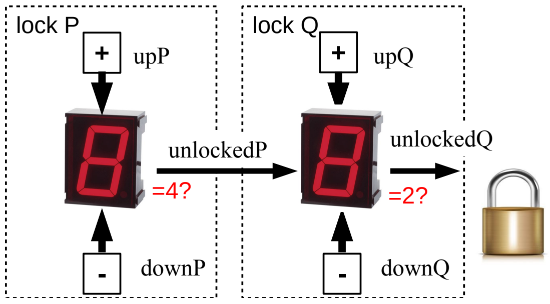


Fig. 1: DDAP system – *SafeLock*

3 DDAP systems

In this paper we focus on Decomposable by Dependency Asynchronous Parallel (DDAP) systems. A DDAP system is composed of two subsystems, running asynchronously in parallel, such that (part of) the inputs of the dependent subsystem are provided by the other subsystem which runs independently. Formally, DDAP systems are defined as follows.

Definition 6 (Dependency). *Given two Kripke structures with inputs* $P = \langle S_P,\ S_P^0,\ IN_P,\ OUT_P,\ T_P,\ \mathcal{L}_P\rangle$ *and* $Q = \langle S_Q,\ S_Q^0,\ IN_Q,\ OUT_Q,\ T_Q,\ \mathcal{L}_Q\rangle$, Q depends on P *if* $OUT_P \cap IN_Q \neq \emptyset$*.*

Definition 7 (DDAP system). *A* DDAP system $\langle P, Q\rangle$ *is a system having two components* P*,* Q *satisfying the following properties:*

- $P = \langle S_P,\ S_P^0,\ IN_P,\ OUT_P, T_P, \mathcal{L}_P\rangle$ *and* $Q = \langle S_Q, S_Q^0, IN_Q, OUT_Q, T_Q, \mathcal{L}_Q\rangle$ *are two Kripke structure with inputs;*
- Q *depends on* P*, but* P *does not depend on* Q*;*
- *only one system at a time is active (interleaving asynchronous parallelism).*

We call $D = OUT_P \cap IN_Q$ the *dependency set* of the DDAP system.

Example 2. Fig. 1 shows an example of DDAP system (called *SafeLock*). The safe lock system is composed by two locks, P and Q, which work in sequence. Both locks have two buttons (`upP` and `downP`, `upQ` and `downQ`) that change the digit of the lock. Lock P becomes *unlocked* (i.e., `unlockedP` $= true$) only if the digit is equal to the stored correct value (in the example, the value 4). Lock Q becomes *unlocked* (i.e., `unlockedQ` $= true$) only if the digit is equal to the stored correct value (in the example, the value 2) and if P is unlocked. So the safe lock is unlocked when Q is unlocked.

Lock P has as inputs $IN_P = \{$`upP`, `downP`$\}$ and as output $OUT_P = \{$`digitP0`, $\ldots$, `digitP9`, `unlockedP`$\}$. It has ten different states $(s_0^p, \ldots, s_9^p)$, distinguished by the value of the digit; $\mathcal{L}_P(s_4^p) = \{$`digitP4`, `unlockedP`$\}$ and $\mathcal{L}_P(s_i^p) = \{$`digitP`$i\}$ for $i = 0, \ldots, 3, 5, \ldots 9$.

Lock Q has as inputs $IN_Q = \{\texttt{upQ}, \texttt{downQ}, \texttt{unlockedP}\}$ and as output $OUT_Q = \{\texttt{digitQ0}, \ldots, \texttt{digitQ9}, \texttt{unlockedQ}\}$. It has eleven different states ($s_0^q, \ldots, s_9^q$, and $\tilde{s}_2^q$), distinguished by the value of the digit and of `unlockedQ`; $\mathcal{L}_Q(\tilde{s}_2^q) = \{\texttt{digitQ2}, \texttt{unlockedQ}\}$, and $\mathcal{L}_Q(s_i^q) = \{\texttt{digitQ}i\}$ for $i = 0, \ldots, 9$.

The output `unlockedP` of lock P is connected to the corresponding input of lock Q, i.e., the dependency set is $D = \{\texttt{unlockedP}\}$.

Definition 8 (DDAP input sequence). *The* input set *of a DDAP system $K = \langle P, Q\rangle$ is the set $IN_K = IN_P \cup (IN_Q \setminus D)$. An* input sequence *for the DDAP K is a sequence $J_0, \ldots, J_n$ such that $J_i \in \mathcal{P}(IN_K)$.*

We define the concept of a trace of a DDAP system, reflecting the fact that only one component makes a move at each step, and that, when the dependent component moves, it reads some of its inputs from the outputs of the independent component.

Definition 9 (DDAP trace). *Given an input sequence $J_0, \ldots, J_n, \ldots$ for a DDAP system $\langle P, Q\rangle$, a* trace *is the sequence (p_0, q_0), J_0, (p_1, q_1), $\ldots$, (p_n, q_n), J_n, (p_{n+1}, q_{n+1}), $\ldots$ such that:*
(1) $p_0 \in S_P^0$ and $q_0 \in S_Q^0$;
(2) $((p_i, J_i \cap IN_P, p_{i+1}) \in T_P \wedge q_i = q_{i+1}) \oplus ((q_i, (J_i \cap IN_Q) \cup (\mathcal{L}_P(p_i) \cap D), q_{i+1}) \in T_Q \wedge p_i = p_{i+1})$.

Requirement (2) specifies that either the component P moves from p_i to p_{i+1} and Q remains still in state $q_i = q_{i+1}$, or component Q moves from q_i to q_{i+1} and P remains still in state $p_i = p_{i+1}$. When Q moves, it reads some of its inputs from the outputs of P (i.e., $\mathcal{L}_P(p_i) \cap D$).

Example 3. For the safe lock system *SafeLock*, the input set is $IN_{SafeLock} = \{\texttt{up}_\texttt{P}, \texttt{down}_\texttt{P}, \texttt{up}_\texttt{Q}, \texttt{down}_\texttt{Q}\}$. Assuming that the both locks are initialized to 0, a possible trace leading to the state in which the global lock is unlocked is: (s_0^p, s_0^q), $\{\texttt{up}_\texttt{P}\}$, (s_1^p, s_0^q), $\{\texttt{up}_\texttt{P}\}$, (s_2^p, s_0^q), $\{\texttt{up}_\texttt{P}\}$, (s_3^p, s_0^q), $\{\texttt{up}_\texttt{P}\}$, (s_4^p, s_0^q), $\{\texttt{up}_\texttt{Q}\}$, (s_4^p, s_1^q), $\{\texttt{up}_\texttt{Q}\}$, $(s_4^p, \tilde{s}_2^q)$.

Note that DDAP systems can be extended to systems with more that two subsystems, as shown in Section 5.

3.1 Encoding DDAP systems in NuSMV

NuSMV permits to split a model in different modules and run several module instances in the *main* module. Modules instances can be run in a synchronous or asynchronous way. Asynchronous modules instances are created through the keyword **process**; at each step, one process is nondeterministically chosen and executed, while the other processes do not run and so do not change their state.

A DDAP system can be easily encoded in NuSMV. Subsystems P and Q are defined as two NuSMV modules, as described in Section 2.2, and asynchronously instantiated in the main module (as processes `procP` and `procQ`), so that only

```
MODULE lockP
DEFINE keyP := 4;
IVAR -- IN_P
   upP: boolean;
   downP: boolean;
VAR -- OUT_P
   digitP: 0 .. 9;
DEFINE -- OUT_P
   unlockedP := digitP = keyP;-- D = OUT_P ∩ IN_Q
ASSIGN
   init(digitP) := 0;
   next(digitP) :=
      case
         upP & !downP: (digitP + 1) mod 10;
         downP & !upP: (digitP + 9) mod 10;
         TRUE: digitP;
      esac;
```

Code 1: Lock P

```
MODULE lockQ
DEFINE keyQ := 2;
IVAR -- IN_Q
   upQ: boolean;
   downQ: boolean;
   unlockedP: boolean; -- D = OUT_P ∩ IN_Q
VAR -- OUT_Q
   digitQ: 0 .. 9;
DEFINE -- OUT_Q
   unlockedQ := digitQ = keyQ & unlockedP;
ASSIGN
   init(digitQ) := 0;
   next(digitQ) :=
      case
         upQ & !downQ: (digitQ + 1) mod 10;
         downQ & !upQ: (digitQ + 9) mod 10;
         TRUE: digitQ;
      esac;
```

Code 2: Lock Q

```
MODULE main
VAR
    procP: process lockP;
    procQ: process lockQ;
TRANS procP.unlockedP = procQ.unlockedP;
```

Code 3: DDAP system *SafeLock*

one subsystem is executed at a time. The connection between P outputs and Q inputs is established by a `TRANS` declaration in which each output x of P belonging to the dependency set (i.e., $x \in OUT_P \cap IN_Q$) is linked with the corresponding input of Q (i.e., `procP.`x = `procQ.`x). In the sequel, we refer to this global model as *whole model.*

Example 4. We have encoded the running case study *SafeLock* in NuSMV. Codes 1 and 2 show the NuSMV modules for locks P and Q; Code 3 shows the main module that asynchronously executes the two locks and connects the output `unlockedP` of P with the corresponding input of Q.

4 Test Generation for DDAP systems

In this section we present a novel technique for test generation by model checking for DDAP systems. The technique introduces an abstraction that exploits the dependency among the subsystems.

Definition 10 (DDAP test). *A* test *for a DDAP system $\langle P, Q\rangle$ is a finite trace (p_0, q_0), J_0, (p_1, q_1), ..., (p_{n-1}, q_{n-1}), J_{n-1}, (p_n, q_n).*

Definition 11 (Soundness). *A test generation method for DDAP systems is* sound *if each produced sequence is a test for the DDAP.*

```
-> State: 1.1 <-
  procP.upP = FALSE
  procP.downP = FALSE
  procP.digitP = 0
  procP.unlockedP = FALSE
  procQ.upQ = FALSE
  procQ.downQ = FALSE
  procQ.unlockedP = FALSE
  procQ.digitQ = 0
  procQ.unlockedQ = FALSE
  procP.keyP = 4
  procQ.keyQ = 2trap
-> Input: 1.2 <-
  _process_selector_ = main
  running = TRUE
  procQ.running = FALSE
  procP.running = FALSE
-> State: 1.2 <-
  procP.upP = TRUE
-> Input: 1.3 <-
  _process_selector_ = procP
  running = FALSE
  procP.running = TRUE
-> State: 1.3 <-
  procP.digitP = 1
-> Input: 1.4 <-
-> State: 1.4 <-
  procP.digitP = 2
-> Input: 1.5 <-
-> State: 1.5 <-
  procP.digitP = 3
-> Input: 1.6 <-
-> State: 1.6 <-
  procP.downP = TRUE
  procP.digitP = 4
  procP.unlockedP = TRUE
  procQ.upQ = TRUE
  procQ.unlockedP = TRUE
-> Input: 1.7 <-
  _process_selector_ = procQ
  procQ.running = TRUE
  procP.running = FALSE
-> State: 1.7 <-
  procQ.digitQ = 1
-> Input: 1.8 <-
-> State: 1.8 <-
  procP.upP = FALSE
  procP.downP = FALSE
  procQ.upQ = FALSE
  procQ.digitQ = 2
  procQ.unlockedQ = TRUE
```

Fig. 2: Witness for the test predicate $\mathbf{F}(\texttt{procQ.unlockedQ})$

If a DDAP system is specified as a *whole model* (as described in Section 3.1), the technique presented in Section 2.3 can be used to generate tests for the DDAP. Let us call this technique M_{whole}.

Definition 12 (Completeness). *A test generation method M for DDAP systems is* complete *if M generates a test suite covering all the* feasible *test predicates of the whole model.*

Theorem 1. *M_{whole} is sound and complete.*

Example 5. Consider the DDAP system *SafeLock* shown in Codes 1, 2, and 3. A test predicate for the system (using M_{whole}) is $\mathbf{F}(\texttt{procQ.unlockedQ})$, requiring that lock Q can become unlocked. In order to find a test for covering the test predicate, we check the trap property $!\mathbf{F}(\texttt{procQ.unlockedQ})$, saying that Q never becomes unlocked. Since the test predicate is feasible, the trap property is violated and the returned counterexample is a witness for the test predicate. The counterexample is shown in Fig. 2; we can see that, in the last state of the sequence, the test predicate is covered because `procQ.unlockedQ` becomes true.

4.1 Decomposition by dependency abstraction

Given a test predicate *tp* over a DDAP system $\langle P, Q\rangle$, if *tp* contains only labels of P, one can apply the cone of influence (COI) abstraction technique [9], by not considering Q, and generating a test only for P. If the test predicate *tp* contains only labels of Q, instead, the application of COI is not effective, since it cannot simplify the model: indeed, P provides input values to Q and so both P and Q must be considered. A basic approach is to generate a test using M_{whole} that, however, may suffer from the state space explosion problem.

Algorithm 1 Test generation algorithm M_{DD}

```
Require: Two specifications $P$ and $Q$
Require: A test predicate $tpQ$ for $Q$
Ensure: A test for the DDAP system
 1: $testQ \leftarrow$ getWitness($tpQ$)
 2: if $testQ \neq$ UNFEASIBLE then
 3:     $inputSeq \leftarrow$ getInputSeq($testQ, P$)
 4:     $rcP \leftarrow$ getLTL($inputSeq$)
 5:     $testP \leftarrow$ getWitness($rcP$)
 6:     if $testP \neq$ UNFEASIBLE then
 7:         return merge($testP, testQ$)
 8:     else
 9:         return UNKNOWN            ▷ It is unknown if the test predicate is feasible
10:     end if
11: else
12:     return UNFEASIBLE                        ▷ The test predicate is unfeasible
13: end if
```

We propose an abstraction that exploits (un)dependency between inputs and outputs to decompose the complete system; the proposed test generation approach consists in generating two tests, one over Q and one P, and merging them later. Alg. 1 shows the test generation algorithm M_{DD} we propose.

Given a test predicate tpQ for Q, if tpQ is feasible, we compute its witness (line 1 in Alg. 1) by asking the model checker for a counterexample for the trap property $\neg tpQ$. The counterexample is a trace of Q

$$testQ = q_0, IQ_0, \ldots, q_m$$

where $q_0 \in S^0_Q$, and $IQ_j \subseteq IN_Q$ is the set of inputs of Q applied at state q_j to obtain state q_{j+1}, $j = 0, \ldots, m-1$. We identify the inputs coming from machine P (those of the dependency set) as $IQ_j \cap D$.

We split the sequence $testQ$ in subsequences σ_i, $i = 0, \ldots, n$, such that atomic propositions of the dependency set remain unchanged:

$$\begin{aligned} testQ &= q_0, \sigma_0, \sigma_1, \ldots, \sigma_n \\ &= q_0, \overbrace{\underbrace{IQ_0, q_1, \ldots, q_{k_1}}_{D_0}}^{\sigma_0}, \overbrace{\underbrace{IQ_{k_1}, q_{k_1+1}, \ldots, q_{k_2}}_{D_1}}^{\sigma_1}, \ldots, \\ &\quad \overbrace{\underbrace{IQ_{k_i}, q_{k_i+1}, \ldots, IQ_{k_{i+1}-1}, q_{k_{i+1}}}_{D_i}}^{\sigma_i}, \ldots, \overbrace{\underbrace{IQ_{k_n}, q_{k_n+1}, \ldots, q_m}_{D_n}}^{\sigma_n} \end{aligned}$$

where $n < m$ and $0 = k_0 < \ldots < k_n < m$, and, for each σ_i, D_i is the set of inputs of the dependency set that are applied all over σ_i, i.e., $\forall j = k_i, \ldots, k_{i+1} - 1$: $IQ_j \cap D = D_i$.

Given the sequence $D_0, \ldots, D_n$ (called *inputSeq* in Alg. 1), we build a reachability condition *rcP* over P as LTL formula (line 4 in Alg. 1), requiring that

$n+1$ subsequent states (not necessarily contiguous) exist, in which P produces the output values D_i requested by Q to start the computation σ_i. It holds

$$rcP = \mathbf{F}\left(\bigwedge_{d_0 \in D_0} d_0 \wedge \mathbf{F}\left(\ldots \mathbf{F}\left(\bigwedge_{d_{n-1} \in D_{n-1}} d_{n-1} \wedge \mathbf{F}\left(\bigwedge_{d_n \in D_n} d_n\right)\right)\ldots\right)\right)$$

If *rcP* is feasible, we compute its witness as counterexample for the trap property $\neg rcP$ (line 5 in Alg. 1)

$$testP = p_0, IP_0, \ldots, p_t$$

P produces the output values D_i in the $n+1$ states p_{h_1}, p_{h_2}, ..., p_{h_n}, p_t. We split the sequence *testP* after states p_0, p_{h_1}, p_{h_2}, ..., p_{h_n}, obtaining the following computation segments

$$\begin{aligned} testP &= p_0, \delta_0, \delta_1, \ldots, \delta_n \\ &= p_0, \overbrace{IP_0, p_1, \ldots, \underbrace{p_{h_1}}_{D_0}}^{\delta_0}, \overbrace{IP_{h_1}, p_{h_1+1}, \ldots, \underbrace{p_{h_2}}_{D_1}}^{\delta_1}, \ldots, \\ &\quad \overbrace{IP_{h_i}, p_{h_i+1}, \ldots, IP_{h_{i+1}-1}, \underbrace{p_{h_{i+1}}}_{D_i}}^{\delta_i}, \ldots, \overbrace{IP_{h_n}, p_{h_n+1}, \ldots, \underbrace{p_t}_{D_n}}^{\delta_n} \end{aligned}$$

with $0 = h_0 < h_1 < \ldots < h_n < t$, and where $p_0 \in S_P^0$ and $IP_j \subseteq IN_P$ is the set of inputs of P applied at state p_j to obtain state p_{j+1}. In the last state $p_{h_{i+1}}$ of a subsequence δ_i, P produces the output values D_i necessary to Q for beginning the subsequence σ_i, i.e., $\mathcal{L}_P(p_{h_{i+1}}) \cap D = D_i$.

The test for the DDAP system can be built, as described below, using information coming from *testP* and *testQ* (line 7 in Alg. 1).

Let $\langle \delta_i \circ q \rangle$ indicate the sequence $IP_{h_i}, (p_{h_i+1}, q), \ldots, (p_{h_{i+1}}, q)$ of the DDAP system in which P executes δ_i and Q keeps still at state q.

Let $\langle \sigma_i \circ p \rangle$ indicate the sequence $IQ_{k_i} \setminus D, (p, q_{k_i+1}), \ldots, (p, q_{k_{i+1}})$ of the DDAP system in which Q executes σ_i and P keeps still at state p.

A test for the DDAP system is the sequence

$$\begin{aligned} testPQ = {} & (p_0, q_0), \langle \delta_0 \circ q_0 \rangle, \langle \sigma_0 \circ p_{h_1} \rangle, \langle \delta_1 \circ q_{k_1} \rangle, \ldots, \\ & \langle \sigma_{i-1} \circ p_{h_i} \rangle, \langle \delta_i \circ q_{k_i} \rangle, \langle \sigma_i \circ p_{h_{i+1}} \rangle, \ldots, \langle \delta_n \circ q_{k_n} \rangle, \langle \sigma_n \circ p_t \rangle \end{aligned}$$

Example 6. In this example we show how to apply the decomposition by dependency abstraction to the safe lock system *SafeLock* presented in Example 2, for generating the test that covers the test predicate $\mathbf{F}(\texttt{unlockedQ})$ in component Q. The test built for Q is

$$testQ = s_0^q, \underbrace{\overbrace{\{\texttt{up}_\texttt{Q}, \texttt{unlockedP}\}, s_1^q, \{\texttt{up}_\texttt{Q}, \texttt{unlockedP}\}, \tilde{s}_2^q}^{\sigma_0}}_{D_0 = \{\texttt{unlockedP}\}}$$

The corresponding reachability condition on P is

$$rcP = \mathbf{F}\,(\texttt{unlockedP})$$

The test built for P is

$$testP = s_0^p, \overbrace{\{\texttt{up}_\texttt{P}\}, s_1^p, \{\texttt{up}_\texttt{P}\}, s_2^p, \{\texttt{up}_\texttt{P}\}, s_3^p, \{\texttt{up}_\texttt{P}\}, s_4^p}^{\delta_0}\underbrace{}_{D_0=\{\texttt{unlockedP}\}}$$

The test for the DDAP system is

$$testPQ = (s_0^p, s_0^q), \overbrace{\{\texttt{up}_\texttt{P}\}, (s_1^p, s_0^q), \{\texttt{up}_\texttt{P}\}, (s_2^p, s_0^q), \{\texttt{up}_\texttt{P}\}, (s_3^p, s_0^q), \{\texttt{up}_\texttt{P}\}, (s_4^p, s_0^q)}^{\langle \delta_0 \circ s_0^q \rangle},$$
$$\underbrace{\{\texttt{up}_\texttt{Q}\}, (s_4^p, s_1^q), \{\texttt{up}_\texttt{Q}\}, (s_4^p, \tilde{s}_2^q)}_{\langle \sigma_0 \circ s_4^p \rangle}$$

Theorem 2 (Soundness). *The test generation method M_{DD} is sound.*

Proof. Proving the soundness of the proposed technique M_{DD} corresponds to prove that *testPQ* is a test for the DDAP system.

testP is a valid trace for P because $\forall j = 0, \ldots, t-1$: $(p_j, IP_j, p_{j+1}) \in T_P$. *testQ* is a valid trace for Q because $\forall j = 0, \ldots, m-1$: $(q_j, IQ_j, q_{j+1}) \in T_Q$. In Def. 9, (1) holds since $p_0 \in S_P^0$ and $q_0 \in S_Q^0$ (by def. of *testP* and *testQ*). In Def. 9, (2) holds for all the transitions of *testPQ*:

- for the initial transition $((p_0, q_0), IP_0, (p_1, q_0))$, because $(p_0, IP_0, p_1) \in T_P$;
- for the subsequent transitions $((p,q), I, (p', q'))$, since one of the following cases occurs:
  - if the transition is in a $\langle \delta_i \circ q_{k_i} \rangle$, then $(p, I, p') \in T_P \wedge q = q' = q_{k_i}$;
  - if the transition is in a $\langle \sigma_i \circ p_{h_{i+1}} \rangle$, then $(q, I \cup D_i, q') \in T_Q \wedge \mathcal{L}_P(p_{h_{i+1}}) \cap D = D_i \wedge p = p' = p_{h_{i+1}}$;
  - if the transition moves from $\langle \delta_i \circ q_{k_i} \rangle$ to $\langle \sigma_i \circ p_{h_{i+1}} \rangle$ (with $i = 0, \ldots, n$), then $(q, I \cup D_i, q') \in T_Q \wedge p = p' = p_{h_{i+1}}$, with $I = IQ_{k_i} \setminus D_i$, $q = q_{k_i}$, and $q' = q_{k_i+1}$;
  - if the transition moves from $\langle \sigma_{i-1} \circ p_{h_i} \rangle$ to $\langle \delta_i \circ q_{k_i} \rangle$ (with $i = 1, \ldots, n$), then $(p, I, p') \in T_P \wedge q = q' = q_{k_i}$, with $I = IP_{h_i}$, $p = p_{h_i}$, and $p' = p_{h_i+1}$.

Proposition 1 (Incompleteness). *The method M_{DD} is not complete.*

Proof. A test predicate in Q may be covered by more than one test; the model checking approach, however, returns only one test. It is easy to build a DDAP system for which, given a test predicate *tpQ* for Q, there exist two tests, *testQ* and *testQ'*, for covering *tpQ* in Q, and such that P can provide the values required by *testQ'*, but not the values required by *testQ*. If the model checker returns *testQ*, the test predicate is not covered with M_{DD} (Alg. 1 returns `UNKNOWN`), although it can be covered using M_{whole}.

```
MODULE P
VAR -- OUT_P
    x: 1 .. 4; -- D
ASSIGN
    x := {2, 4};
```

```
MODULE Q
IVAR -- IN_Q
    x: 1 .. 4; -- D
DEFINE
    y := (x + 1) mod 3;
```

```
MODULE main
VAR
    procP: process P;
    procQ: process Q;
TRANS procP.x = procQ.x;
```

Code 4: Example of DDAP system for proving that M_{DD} is not complete

Let us consider the DDAP system shown in Code 4. In order to cover the test predicate $\mathbf{F}(\mathtt{y} = 2)$ in Q, M_{DD} can require by P either value 1 or value 4 for the input variable `x`. If the required value is 1, M_{DD} can not find a test on P to provide 1 as value for `x`, since `x` can only assume values 2 and 4 in P; so M_{DD} returns `UNKNOWN`. However, the test predicate can be covered with M_{whole} (which finds a witness for the corresponding test predicate $\mathbf{F}(\mathtt{procQ.y} = 2)$), and could be covered using M_{DD} as well, if Q would request 4 as input value for `x`.

5 Generalization to DDAP systems with *n* components

DDAP systems (see Def. 7) can be extended to systems with more than two components.

Definition 13 (n-DDAP system). *An* n-DDAP system *is a system having n components $C_1, \ldots, C_n$ (with $n \geq 2$) satisfying the following properties:*

- *$C_i = \langle S_{C_i}, S^0_{C_i}, IN_{C_i}, OUT_{C_i}, T_{C_i}, \mathcal{L}_{C_i} \rangle$ is a Kripke structure with inputs;*
- *C_i depends only on C_{i-1}, for each $i = 2, \ldots, n$; C_1 does not depend on any other component;*
- *only one system at a time is active (interleaving asynchronous parallelism).*

We can adapt the test generation approach presented in Alg. 1 for dealing with n-DDAP systems. Alg. 2 shows the modified algorithm M^n_{DD}. Given a test predicate for a component C_i, the algorithm builds a test for C_i (line 1). Then, if the test predicate is feasible, for each previous component C_j it computes the reachability condition that specifies the values that C_j must pass to C_{j+1} (lines 5 and 6). If a test satisfying the reachability condition can be built for C_j, such test is merged with the previous tests generated so far for components $C_{j+1}, \ldots, C_i$ (line 9). If for a component C_j the test cannot be built, M^n_{DD} returns the `UNKNOWN` result, otherwise, at the end, it returns a test for the n-DDAP system.

6 Initial Experiment

We run all the experiments on a Linux machine, Intel(R) Core(TM) i7 CPU, 4 GB RAM. We have developed NuSMV models in the NuSeen framework[3] which

[3] `https://code.google.com/a/eclipselabs.org/p/nuseen/`

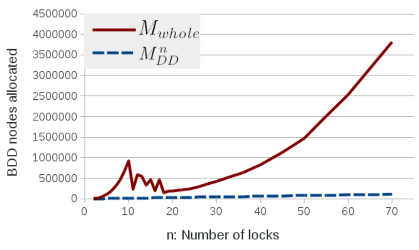
Algorithm 2 Test generation algorithm M^n_{DD} for n-DDAP systems

```
Require: An $n$-DDAP system $\{C_1, \dots, C_n\}$
Require: A test predicate $tp$ for a $C_i$
Ensure: A test for the $n$-DDAP system
 1: $componentTest \leftarrow$ getWitness($tp$)
 2: $systemTest \leftarrow componentTest$
 3: if $componentTest \neq$ UNFEASIBLE then
 4:     for $j = i - 1, \dots, 1$ do
 5:         $inputSeq \leftarrow$ getInputSeq($componentTest, C_j$)
 6:         $rc \leftarrow$ getLTL($inputSeq$)
 7:         $componentTest \leftarrow$ getWitness($rc$)
 8:         if $componentTest \neq$ UNFEASIBLE then
 9:             $systemTest \leftarrow$ merge($componentTest, systemTest$)
10:         else
11:             return UNKNOWN          ▷ It is unknown if the test predicate is feasible
12:         end if
13:     end for
14: else
15:     return UNFEASIBLE                      ▷ The test predicate is unfeasible
16: end if
17: return $systemTest$
```

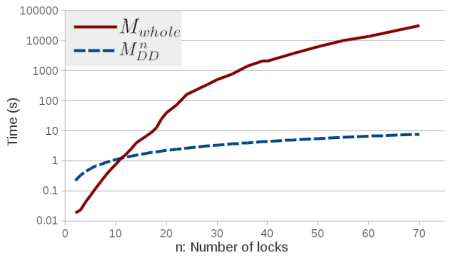
provides an interface to the NuSMV model checker [6] and to a model advisor for NuSMV specifications [4].

We have experimented our approach on the n-DDAP system *n-SafeLock*, an extension of the DDAP system *SafeLock* described in Example 2. *n-SafeLock* is composed of n locks L_1, ..., L_n, such that each lock L_i is unlocked if it contains the correct digit and (except for L_1) if the previous lock is unlocked. We have applied the basic technique M_{whole} and the proposed technique M^n_{DD} (see Alg. 2) on different instances of *n-SafeLock*, using an increasing number of locks n. For each experiment, we have always tried to cover the test predicate $\mathbf{F}$(`unlockedLn`) over the last lock L_n; all the test predicates are feasible. M_{whole} has find a test for each test predicate (as expected from Thm. 1); also M^n_{DD} has always obtained a test for the whole system (no `UNKNOWN` result).

Fig. 3 shows the experimental results. Fig. 3a shows the memory consumption (in terms of number of BBD nodes allocated) of the test generation using M_{whole} and using M^n_{DD}; we can see that, using M_{whole}, the required memory grows exponentially, whereas, using M^n_{DD}, it grows linearly. Fig. 3b shows the time taken by the two test generation methods (using a logarithmic); the required time grows exponentially using M_{whole}, whereas it grows linearly using M^n_{DD}. Note that M_{whole} calls the model checker only once, while M^n_{DD} does it n times. For small values of n, when the instantiation time constitutes the main part of the execution time, M_{whole} outperforms M^n_{DD}.



(a) Memory



(b) Time

Fig. 3: Experiment results

7 Related work

In [2] we have proposed a test generation technique for *sequential nets* 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. Apart from the different notation, that technique shares with M_{DD} the fact that the generation of the tests is performed for the single subsystems that are subsequently combined. However, that technique only supports sequential systems, whereas M_{DD} supports interleaving asynchronous systems.

The technique presented in [2] has been extended in [3] for handling the passing of information between subsystems, in a similar way as done in M_{DD} with the dependency set. However, since the subsystems run in sequence, the information between two subsystems P and Q can only be passed by P to Q at the end of a P run in order to start a Q run; in M_{DD}, instead, P can pass information to Q several times in different states of their traces.

With respect to [2,3], the technique proposed here has required to handle, as test predicates, LTL temporal formulae. Moreover, tests are no more built by concatenating the tests for the single components, but by merging them.

Since our approach is based on model checking, we mainly relate to abstraction techniques for formal verification. The *cone of influence* (COI) technique [9] 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 [18] 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. The *data abstraction* technique [9], 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 [7] a technique to iteratively refine an abstract model is presented. 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.

A technique for sequential *modular* decomposition for property verification of complex programs is presented in [17]. 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 guarantee that the property holds in *any* state, while in our approach, since we want to find only a counterexample, we only need to find a path leading to interesting states.

An approach performing test generation by decomposing sequential *programs*, called SMART, is presented in [13]. 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 (like M_{whole}), while our approach is 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.

8 Conclusions

We have proposed a test generation approach by model checking for Decomposable by Dependency Asynchronous Parallel (DDAP) systems, i.e., systems composed by several subsystems connected together in a way that (part of) the inputs of one subsystem are provided by another subsystem. The approach is based on a decompositional abstraction: It builds tests for the single subsystems and combines them later in order to obtain a global system test. Such approach permits to mitigate the state explosion problem of model checking. The method has been proved to be sound but not complete.

As future work, we plan to apply the proposed technique M_{DD} to more complex systems, possibly leading to `UNKNOWN` results. This would require to improve M_{DD} to achieve its completeness, on the base of the following intuition.

When Alg. 1 returns the `UNKNOWN` result, it means that the value requested by test *testQ* cannot be provided by P. In this case, the technique could ask for another test $testQ'$ for Q, and check if now the values requested by $testQ'$ can be provided by P; such procedure should be iterated until a test for the whole system is returned or no new test on Q can be found.

The approach M_{DD} is suitable for building tests for test predicates defined over AP_Q, i.e., the labels of only Q (or of only a component in an $n-$DDAP system). As future work, we plan to extend the technique for handling *general* test predicates built over all the labels of the system, i.e., over $AP_P \cup AP_Q$.

A further improvement could be dealing with systems in which one component may depend on several components. In this case, the dependency relation would be represented by an acyclic graph.

References

1. 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.
2. 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.
3. P. Arcaini and A. Gargantini. Test Generation for Sequential Nets of Abstract State Machines with Information Passing. *Science of Computer Programming*, (0):–, 2014.
4. P. Arcaini, A. Gargantini, and E. Riccobene. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering*, 7(2):97–107, 2011.
5. M. C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. In *Proceedings, Symposium on Logic in Computer Science (LICS), 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 260–266. IEEE Computer Society, 1986.
6. 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 *LNCS*. Springer, July 2002.
7. 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.
8. 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.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
10. 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.

11. G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.
12. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162, London, UK, 1999. Springer Berlin Heidelberg.
13. 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.
14. R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.
15. B. Josko. A context dependent equivalence relation between kripke structures. In E. Clarke and R. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 204–213. Springer Berlin Heidelberg, 1991.
16. 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.
17. 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.
18. 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.
19. W. Prenninger and A. Pretschner. Abstractions for Model-Based Testing. *Electron. Notes Theor. Comput. Sci.*, 116:59–71, Jan. 2005.
20. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2006.
21. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.