Chaining Test Cases for Reactive System Testing*

(extended version)

Peter Schrammel, Tom Melham, and Daniel Kroening

University of Oxford
Department of Computer Science
first.lastname@cs.ox.ac.uk

Abstract. Testing of synchronous reactive systems is challenging because long input sequences are often needed to drive them into a state to test a desired feature. This is particularly problematic in *on-target testing*, where a system is tested in its real-life application environment and the amount of time required for resetting is high. This paper presents an approach to discovering a *test case chain*—a single software execution that covers a group of test goals and minimises overall test execution time. Our technique targets the scenario in which test goals for the requirements are given as safety properties. We give conditions for the existence and minimality of a single test case chain and minimise the number of test case chains if a single test case chain is infeasible. We report experimental results with a prototype tool for C code generated from Simulink models and compare it to state-of-the-art test suite generators.

1 Introduction

Safety-critical embedded software, e.g., in the automotive or avionics domain, is often implemented as a *synchronous reactive system*. These systems compute their new state and their output as functions of old state and the given inputs. As these systems frequently have to satisfy high safety standards, tool support for systematic testing is highly desirable. The completeness of the testing process is frequently measured by defining a set of *test goals*, which are typically formulated as reachability properties. A good-quality test suite is a set of input sequences that drive the system into states that cover a large fraction of those goals.

Test suites generated by random test generators often contain a huge number of redundant test cases. Directed test case generation often requires lengthy input sequences to drive the system into a state where the desired feature can be tested. Furthermore, to execute the test suite, test cases must be chained manually or the system must be reset after executing each test case. This is a serious problem in

* Supported by the EU FP7 STREP PINCETTE, the ARTEMIS VeTeSS project, and ERC project 280053.

```
void init(t_state *s) { s->mode = OFF; s->speed = 0; s->enable = FALSE; }
void compute(t_input *i, t_state *s) {
  mode = s->mode;
  switch(mode) {
    case ON: if(i->gas || i->brake) s->mode=DIS; break;
    case DIS:
      if( (s->speed==2 && (i->dec || i->brake)) || (s->speed==0 && (i->acc || i->gas)) )
        s->mode=ON;
      break;
    case OFF:
      if( s->speed==0 && s->enable && (i->gas || i->acc) ||
          s->speed==1 && i->button ||
          s->speed==2 && s->enable && (i->brake || i->dec) )
        s->mode=ON;
      break;
  }
  if(i->button) s->enable = !s->enable;
  if((i->gas || mode!=ON && i->acc) && s->speed<2) s->speed++;
  if((i->brake || mode!=ON && i->dec) && s->speed>0) s->speed--;
}
```

Fig. 1. Code generated for cruise controller example

on-target testing, where a system is tested in its real-life application environment and resetting might be very time-consuming [1].

This paper presents an approach to discovering a *test case chain*—a single test case that covers a set of multiple test goals and minimises overall test execution time. The essence of the problem is to find a shortest path through the system that covers all the test goals.

Example. To illustrate the problem and our approach, we reuse the classical cruise controller example given in [2]. There are five Boolean inputs, two for actuation of the *gas* and *brake* pedals, a toggle *button* to enable the cruise control, and two sensors indicating whether the car is *acc*- or *dec*elerating. There are three state variables: *speed*, *enable*, which is true when cruise control is enabled, and *mode* indicating whether cruise control is turned *OFF*, actually active (*ON*), or temporarily inactive, i.e., *DIS*engaged while user pushes the gas or brake pedal. A C implementation, with the structure typical of code generated from SIMULINK models, is given in Fig. 1 and its state machine is depicted in Fig. 2. The function `compute` is executed periodically (e.g. on a timer interrupt). Thus, there is a notion of *step* that relates to execution time.

We formulate some LTL properties for which we want to generate test cases:

p_1: $\mathbf{G}\big(mode = ON \wedge speed = 1 \wedge dec \Rightarrow \mathbf{X}(speed = 1)\big)$
p_2: $\mathbf{G}\big(mode = DIS \wedge speed = 2 \wedge dec \Rightarrow \mathbf{X}(mode = ON)\big)$
p_3: $\mathbf{G}\big(mode = ON \wedge brake \Rightarrow \mathbf{X}(mode = DIS)\big)$
p_4: $\mathbf{G}\big(mode = OFF \wedge speed = 2 \wedge \neg enable \wedge button \Rightarrow \mathbf{X}\ enable\big)$

We observe that each of the properties above relates to a particular transition in the state machine (shown as bold edge labels in Fig. 2). A *test case* is a sequence of inputs that determines a (bounded) execution path through the system. The *length* of a test case is the length of this sequence. A test case *covers a property* if it triggers the transition the property relates to. A *test suite* is a set of test cases that covers all the properties.

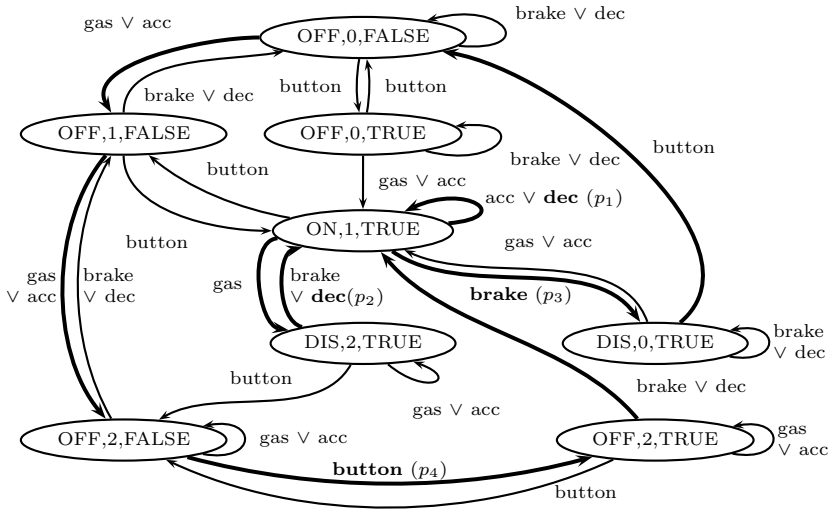

Fig. 2. State machine of the example. Edges are labelled by inputs and nodes by state $\langle mode, speed, enable \rangle$. Properties are in bold, bold edges show a minimal test case chain.

Ideally, we can obtain a single test case that covers all properties in a single execution. We call a test case that covers a sequence of properties a *test case chain*. Our goal is to synthesise minimal test case chains—test case chains with fewest transitions. It is not always possible to generate a single test case chain that covers all properties; multiple test case chains may be required.

We compute such a minimal test case chain from a set of start states I via a set of given properties $P = \{p_1, p_2, \ldots\}$ to a set of final states F. For our example, with $I = F = \{mode = OFF \wedge speed = 0 \wedge \neg enable\}$ and $P = \{p_1, p_2, p_3, p_4\}$, for instance, we obtain the test case chain consisting of the bold edges in Fig. 2. First, this chain advances to p_4, then covers p_1, p_2, and p_3, and finally goes to F. One can assert that this path has the minimal length of 9 steps.

Testing problems similar to ours have been addressed by research on *minimal checking sequences* in conformance testing [3,4,1,5,6]. This work analyses automata-based specifications that encode system control and have transitions labelled with operations on data variables. The challenge here is to find short transition paths based on a given coverage criterion that are feasible, i.e. consistent with the data operations. Random test case generation can then be used to discover such a path. In contrast, our approach analyses the code generated from models or the implementation code itself, and it can handle partial specifications expressed as a collection of safety properties. A common example is acceptance testing in the automotive domain. Our solution uses bounded model checking to generate test cases guaranteed to exercise the desired functionality.

Contributions. The contributions of this paper can be summarised as follows:

– We present a new algorithm to compute minimal test chains that first constructs a weighted digraph abstraction using a reachability analysis, on which the minimisation is performed as a second step. The final step is to compute

the test input sequence. We give conditions for the existence and minimality of a single test case chain and propose algorithms to handle the general case.
- We have implemented a tool, ChainCover[1], for C code generated from Simulink models, on top of the Cbmc bounded model checker and the Lkh travelling salesman problem solver.
- We present experimental results to demonstrate that our approach is viable on a set of benchmarks, mainly from automotive industry, and is more efficient than state-of-the-art test suite generators.

2 Preliminaries

Program model. A *program* is given by (Σ, Υ, T, I) with finite sets of states Σ and inputs Υ, a transition relation $T \subseteq (\Sigma \times \Upsilon \times \Sigma)$, and a set of initial states $I \subseteq \Sigma$. An *execution* of a program is a (possibly) infinite sequence of transitions $s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} s_2 \rightarrow \ldots$ with $s_0 \in I$ and for all $k \geq 0$, $(s_k, i_k, s_{k+1}) \in T$.

Properties. We consider specifications given as a set of safety properties $P = \{p_1, \ldots, p_{|P|}\}$. The properties are given as a formula over state variables s and input variables i and are of the form $\mathbf{G}\big(\varphi \Rightarrow \psi\big)$ where φ describes an *assumption* and ψ is the *assertion* to be checked. φ specifies a test goal, whereas ψ defines the test outcome; hence, for test case generation, only φ is needed. We denote by Π the set of property assumptions. φ is a temporal logic formula built using the operators $\wedge, \vee, \neg, \mathbf{X}$, i.e., it describes sets of finite paths. An execution $\pi = \langle s_0, s_1, \ldots \rangle$ *covers* a property iff it contains a subpath $\langle s_k, \ldots s_{k+j} \rangle$ that satisfies φ (j is the nesting depth of $\mathbf{X}$ operators in φ), i.e.,

$$\exists k \geq 0 : \exists i_k, \ldots, i_{k+j} : \varphi(s_k, i_k, \ldots, s_{k+j}, i_{k+j}) \wedge \bigwedge_{k \leq m \leq k+j} T(s_m, i_m, s_{m+1}).$$

We call the set of states s_k satisfying φ the *trigger* $\widehat{\varphi}$ of the property.

For our method, it is not essential whether φ describes a set of paths or just a set of states; thus, to simplify the presentation, we assume that the property assumptions do not contain $\mathbf{X}$ operators. Single-step transition properties $\mathbf{G}\big(\varphi \Rightarrow \mathbf{X}\psi\big)$ fall into this category, for example. In this case, φ is equivalent to its trigger $\widehat{\varphi}$.

Moreover, we assume that property assumptions are non-overlapping, i.e. the sub-paths satisfying the assumptions do not share any edges. Our minimality results only apply to such specifications. Detecting overlappings is a hard problem [7] that goes beyond the scope of this paper.

Test cases. A *test case* is an input sequence $\langle i_0, \ldots, i_n \rangle$ and generates an execution $\pi = \langle s_0, \ldots, s_{n+1} \rangle$. A test case *covers* a property p iff its execution covers the property.

[1] `http://www.cprover.org/chaincover/`

3 Chaining Test Cases

The problem. We are given a program (Σ, Υ, T, I), properties P, and a set of final states $F \subseteq \Sigma$. A *test case chain* χ is a test case $\langle i_0, \ldots, i_n \rangle$ that covers all properties in P, i.e., its execution $\langle s_0, \ldots, s_{n+1} \rangle$ starts in $s_0 \in I$, ends in $s_{n+1} \in F$, and covers all properties in P. A *minimal test case chain* is a test case chain of minimal length. The final states F are used to ensure the test execution ends in a desired state, e.g. "engines off" or "gear locked in park mode".

Our approach. We now describe our basic algorithm, which has three steps:
(1) *Abstraction:* We construct a *property K-reachability graph* of the system. This is a weighted, directed graph with nodes representing the properties and edges labelled with the number of states through which execution must pass, up to length K, between the properties.
(2) *Optimisation:* We determine the shortest path that covers all properties in the abstraction.
(3) *Concretisation:* Finally, we compute the corresponding concrete test case chain along the abstract path.
We discuss the conditions under which we obtain the *minimal* test case chain. This algorithm is given as Alg. 1.

Algorithm 1: Compute test case chain

Input: program (Σ, Υ, T, I), properties P, formulas I, F, reachability bound K
Output: test case chain $\chi = \langle i_0, \ldots, i_N \rangle$
1 $G = \mathit{BuildPropKReachGraph}(P, I, F, T, K)$
2 $\pi = \mathit{GetShortestPath}(G, I, F)$
3 $\chi = \mathit{GetChain}(G, \pi, T)$
4 **return** χ

3.1 Abstraction: Property K-Reachability Graph

The *property K-reachability graph* is an abstraction of the original program by a weighted, directed graph (V, E, W), with
- vertices $V = \Pi \cup \{I, F\}$, all defining property assumptions, including formulas describing the sets I and F,
- edges $E \subseteq E_{target} \subset V \times V$, as explained below, and
- an edge labelling $W : E \to \mathbb{N}$ assigning to each $(\varphi, \varphi') \in E$ the minimal number of steps bounded by K needed to reach some state satisfying φ' from any state satisfying φ according to the program's transition relation T.

Fig. 3 shows the property 2-reachability graph for our example.

Graph construction. The graph is constructed by the function *BuildPropKReachGraph* (Alg. 2). The main work is done by the function *GetKreachEdges*

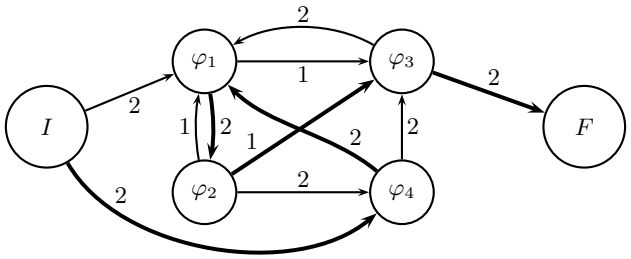

Fig. 3. Test case chaining: property K-reachability graph (for $K = 2$) and minimal test case chain of length $n = 9$ (bold edges) for our example (Fig. 2).

$((V, E, W), T, E_{target}, k)$, which computes the subset of edges E_k that have weight k in the set of interesting edges E_{target}. The constructed graph contains an edge (φ, φ') with weight k iff for the two properties with assumptions φ and φ', a state in φ' is reachable from a state φ in $k \leq K$ steps, and k is the minimal number of steps for reaching φ' from φ. We stop the construction of the graph if a path has been found (line 5). *ExistsPath* is explained below. If we fail to find a path before reaching a given reachability bound K, or there is no path although the graph contains all edges in E_{target}, then we abort (line 6).

Algorithm 2: *BuildPropKReachGraph*

Input: property assumptions Π, formulas I, F, transition function T, reachability bound K
Output: weighted, directed graph (V, E, W)

1 $V \leftarrow \Pi \cup \{I, F\}$
2 $E \leftarrow \emptyset$, $W \leftarrow \emptyset$
3 $E_{target} \leftarrow \left(\bigcup_{\varphi_j \in \Pi} \{(I, \varphi_j), (\varphi_j, F)\}\right) \cup \{(\varphi_j, \varphi_k) \mid \varphi_j, \varphi_k \in \Pi, j \neq k\}$
4 $k \leftarrow 0$
5 **while** $\neg$*ExistsPath*$((V, E, W), I, F)$ **do**
6 **if** $k > K \vee E_{target} = \emptyset$ **then abort** "no chain found for given bound K"
7 **let** $E_k =$ *GetKreachEdges*$((V, E, W), T, E_{target}, k)$
8 $E \leftarrow E \cup E_K$, $E_{target} \leftarrow E_{target} \setminus E_k$
9 **for all** $e \in E_k$ **do** $W \leftarrow W \cup \{e \mapsto k\}$
10 $k \leftarrow k + 1$
11 **return** (V, E, W)

Existence of a covering path. Alg. 2 requires to check for the existence of a covering path (function *ExistsPath*) in each iteration. The existence of a covering path can be formulated as a reachability problem in a directed graph:

Lemma 1. *Let (V, E) be a directed graph of the kind described above. Then, there is a covering path from I to F iff*
(1) all vertices are reachable from I,

(2) F is reachable from all vertices, and
(3) for all pairs of vertices $(v_1, v_2) \in (V \setminus \{I, F\})^2$,
(a) v_2 is reachable from v_1 or (b) v_1 is reachable from v_2.

Proof. In the transitive closure (V, E') of (V, E), v_2 is reachable from v_1 iff there exists an edge $(v_1, v_2) \in E'$.

($\Longrightarrow$): conditions (1) and (2) are obviously necessary. Let us assume that we have a covering path π and there are vertices (v_1, v_2) which neither satisfy (3a) nor (3b). Then neither $\langle v_1, \ldots, v_2 \rangle$ nor $\langle v_2, \ldots, v_1 \rangle$ can be a subpath of π, which contradicts the fact that π is a covering path.

($\Longleftarrow$): Any vertex is reachable from I (1), so let us choose v_1. From v_1 we can reach another vertex v_2 (3a), or, at least, v_1 is reachable from another vertex v_2 (3b), but in the latter case, since v_2 is reachable from I, we can go first to v_2 and then to v_1. Induction step: Let us assume we have a path $\langle I, v_1, \ldots, v_k \rangle$. If there is a vertex v' that is reachable from v_k (3a) we add it to our current path π. If v' is unreachable from v_k, then by (3b), v_k must be reachable from v', and there is a $v_i, i < k$ in $\pi = \langle I, \ldots, v_k \rangle$ from which it is reachable and in this case we obtain the path $\langle I, \ldots, v_i, v', v_{i+1}, \ldots, v_k \rangle$; if there is no such v_i then, at last by (1), v' is reachable from I, so we can construct the path $\langle I, v', \ldots, v_k \rangle$. F is reachable from any vertex (2), thus, we can complete the covering path as soon as all other vertices have been covered. □

Reachability can be checked in constant time on the transitive closure of the graph. Hence, the overall existence check has complexity $\mathcal{O}(|V|^3)$.

3.2 Optimisation: Shortest Path Computation

The next step is to compute the shortest path (function *GetShortestPath* in Alg. 1) covering all nodes in the property K-reachability graph. Such a path is not necessarily Hamiltonian; revisiting nodes is allowed. However, we can compute the transitive closure of the graph using the Floyd-Warshall algorithm [8] (which preserves minimality), and then compute a Hamiltonian path from I to F. If we do not have a Hamiltonian path solver, we can add an edge from F to I and pass the problem to an *asymmetric travelling salesman problem* (ATSP) solver (referred to as *SolveATSP* in the sequel) that gives us the shortest circuit that visits all vertices exactly once. We cut this circuit between F and I to obtain the shortest path π.

Lemma 2 (Minimum covering path). *Let (V, E', W') be the transitive closure of a weighted directed graph (V, E, W), and $I, F \in V$. Then, SolveATSP $(V, E' \cup \{(F, I)\}, W' \cup \{(F, I) \mapsto 1\})$ returns a permutation $\pi = \langle v_0, \ldots, v_{|V|-1} \rangle$ of vertices V such that $\langle v_i = I, \ldots, v_{|V|-1}, v_0, \ldots, v_{i-1} = F \rangle$ is a minimum covering path from I to F.*

Proof. (V, E, W) has a covering path $\langle \ldots, v, v', v, v'', \ldots \rangle$ that is non-Hamiltonian, then (V, E', W') has a Hamiltonian path $\langle \ldots, v, v', v'', \ldots \rangle$ because v'' is reachable from v'.

Any Hamiltonian circuit $\langle v_0, \ldots, v_{|V|-1}\rangle$ returned by *SolveATSP* must contain the edge $(v_i, v_{(i+1) \mod |V|}) = (F, I)$ because (F, I) is the only (and hence the cheapest) edge for reaching I from F.

The obtained path has minimum length because the transitive closure preserves optimality ($W(v_1, v_2) + W(v_2, v_3) = W(v_1, v_3)$). □

For our example, the shortest path has length 9, given as bold edges in Fig. 3.

3.3 Concretisation: Computing the Test Case Chain

Once we have found a minimum covering path π in the property K-reachability graph abstraction, we have to compute the inputs corresponding to it in the concrete program. This is done by the function *CheckPath*(π, T, W) which takes an abstract path $\pi = \langle \varphi_1, \ldots, \varphi_{|V|}\rangle$ and returns the input sequence $\langle i_0, \ldots, i_n\rangle$ corresponding to a concrete path with the reachability distances between each $(\varphi_j, \varphi_{j+1}) \in \pi$ given by the edge weights $W(\varphi_j, \varphi_{j+1})$. Typically, *CheckPath* involves constraint solving; we will discuss our implementation in §5. Hence, *GetChain* in Alg. 1 corresponds to a call to *CheckPath*(π, T, W) and returning the obtained input sequence.

For our example, we obtain, for instance, the sequence ⟨*gas, acc, button, dec, dec, gas, dec, brake, button*⟩ corresponding to the bold edges in Fig. 2.

3.4 Optimality

Since the (non-)existence or the optimality of a chain in the K-reachability abstraction does not imply the (non-)existence or the optimality of a chain in the concrete program, the success of this procedure can only be guaranteed under certain conditions, which we now discuss.

Lemma 3 (Single-state property triggers). *If (1) the program and the properties admit a test case chain, (2) all triggers $\widehat{\varphi}$ of properties in P are singleton sets, and (3) the test case chain χ computed by Alg. 1 visits each property once, then the test case chain is minimal.*

Proof. If each property is visited once, it is guaranteed that the abstract path contains only edges that correspond to concrete paths of minimal length, and hence the test case chain $\chi$is optimal for the concrete program. Otherwise, for a subpath $(\varphi, \varphi', \varphi, \varphi'')$, there might exist an edge (φ', φ'') with $W(\varphi', \varphi'') < W(\varphi', \varphi) + W(\varphi, \varphi'')$ that is only discovered for higher values of K. □

For finite state systems, there is an upper bound for K, the reachability diameter [9,10] beyond that we will not discover shorter pairwise links.

Definition 1 (Reachability diameter). *The reachability diameter d of a system (Σ, Υ, T, I) is the maximum (finite) length of a path in the set of shortest paths between any pair of states $s_i, s_j \in \Sigma$.*

Theorem 1 (Minimal test case chain). *Let d be the reachability diameter of the program, then there is a $K \leq d$ such that, under the preconditions (1) and (2) of Lem. 3, the test case chain χ computed by Alg. 1 is minimal.*

Proof. For $K = d$, it is guaranteed that the abstract path contains only edges of minimal length, and hence the chain is optimal w.r.t the concrete program (even if properties are revisited).

In practice, we can stop the procedure if a chain of acceptable length is found, i.e. we do not compute the reachability diameter but use a user-supplied bound.

4 Generalisations

We will now generalise our algorithm in three ways:

- *Multi-state property triggers:* Dropping the assumption that triggers are single-state may make the concretisation phase fail. Under certain restrictions, we will still find a test case chain if one exists, but we lose minimality.
- Without these restrictions, we might even lose completeness, i.e., the guarantee to find a chain if one exists. We propose two methods to *ensure completeness* under these circumstances: (1) an abstraction refinement that can be used with any ATSP solver, and (2) a method based on restricting the optimisation problem using path constraints that requires a more general solver, e.g. an Answer Set Programming (ASP) solver.
- *Multiple chains:* Dropping the assumption about the existence of a single chain raises the problem of how to generate multiple chains.

4.1 Multi-State Property Triggers

In practice, many properties are multi-state, i.e. preconditions (2) of Lem. 3 is not met. In this case, the abstract covering path might be infeasible in the concrete program, and hence, the naive concretisation of §3.3 might fail. We have to extend the concretisation step to fix such broken chains.

Example 1 (Broken chain). Let us consider the following broken chain in our example with the properties:

$p_1 : \mathbf{G}\big(mode = OFF \wedge \neg enable \wedge button \Rightarrow \mathbf{X}\ enable\big)$

$p_2 : \mathbf{G}\big(mode = ON \wedge brake \Rightarrow \mathbf{X}(mode = DIS)\big)$

with $I = F = \{mode = OFF \wedge speed = 0 \wedge \neg enable\}$.

We obtain a shortest covering path $\langle I, \varphi_1, \varphi_2, F\rangle$ in the abstraction with weights $W(I, \varphi_1) = 0$, $W(\varphi_1, \varphi_2) = 1$, and $W(\varphi_2, F) = 2$. However, Fig. 2 tells us that the path $\langle I, \varphi_1, \varphi_2\rangle$ is not feasible in a single step, but requires two steps, as illustrated in Fig. 4.

A broken chain contains an infeasible subpath $\mathit{failed\_path} = \langle \varphi_1, \ldots, \varphi_k\rangle$ of the abstract path π that involves at least three vertices, such as $\langle I, \varphi_1, \varphi_2\rangle$ in our example above. We extend the concretisation step (*GetChain*) with a

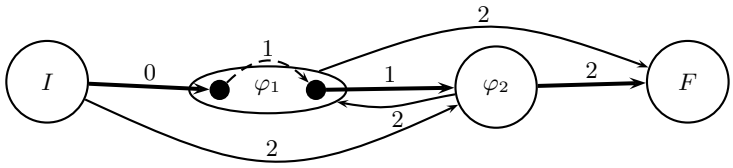

Fig. 4. Broken chain: the path $\langle I, \varphi_1, \varphi_2 \rangle$ is not feasible in a single step, but requires two steps.

chain repair capability. The function *RepairPath* as shown in Alg. 3 iteratively repairs broken chains by incrementing the weights associated with the edges of *failed_path* and checking feasibility of this "stretched" path. We give more details about our implementation in §5.

Algorithm 3: *GetChain* with chain repair

Input: weighted, directed graph (V, E, W), path π, transition relation T
Output: test case chain $\chi = \langle i_0, \ldots, i_N \rangle$

```
1 (feasible, χ, failed_path) ← CheckPath(π, T, W)
2 if feasible then return χ
3 else
4     (succeeded, W, _) ← RepairPath(failed_path, T, W)
5     if ¬succeeded then abort "no chain found for given bound K"
6     (_, χ, _) ← CheckPath(π, T, W)
7     return χ
```

Example 2 (Repaired chain). For the broken chain in our previous example, we will check whether $\langle I, \varphi_1, \varphi_2 \rangle$ is feasible with $W(\varphi_1, \varphi_2)$ incremented by one. This makes the path feasible and we obtain the chain $\chi = \langle$*button,gas,brake,button*$\rangle$.

Completeness. The chain repair succeeds if the given path π admits a chain in the concrete program. In particular, this holds when the states in each property trigger are strongly connected:

Theorem 2 (Multi-state strongly connected property). *If for each property trigger $\widehat{\varphi}$ the states are strongly connected and there exists a test case chain then Alg. 1 (with Alg. 3) will find it.*

In practice, many reactive systems are, apart from an initialisation phase, strongly connected—but, as stressed above, the test case chain might not be minimal.

4.2 Ensuring Completeness

If the shortest path π in the abstraction does not admit a chain in the concrete program, Alg. 1 with chain repair (Alg. 3) will fail to find a test case chain even though one exists, i.e., it is not complete.

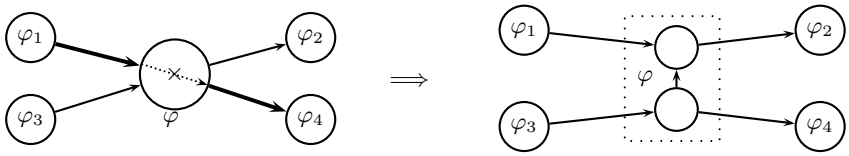

Fig. 5. Abstraction refinement for a failed path $\langle \varphi_1, \varphi, \varphi_4 \rangle$ (bold arrows).

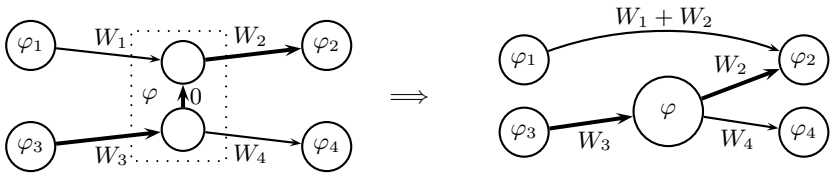

Fig. 6. Collapsing the property refinement group (box) in the refined abstraction to a TSP problem w.r.t. a solution path (bold arrows).

Example 3 (Chain repair fails). In Fig. 4, we have found the shortest abstract path $\langle I, \varphi_1, \varphi_2, F \rangle$. Now assume that the right state in φ_1 is not reachable from the left state. Then the chain repair fails. In this case, there might still be a (non-)minimal path in the abstraction that admits a chain: in our example in Fig. 4, assuming that the left state in φ_1 is reachable from I via φ_2 and F is reachable from the left state in φ_1, we have the feasible path $\langle I, \varphi_2, \varphi_1, F \rangle$.

Abstraction refinement. To obtain completeness in this situation, we propose the following abstraction refinement method shown in Alg. 4. Suppose the chain repair of a covering path π failed with *failed_path* $= \langle \varphi_1, \varphi, \varphi_4 \rangle$ (*succeeded* = *false* in line 5).

1. We refine the graph by splitting vertex φ in *failed_path* as illustrated in Fig. 5 that rules out the infeasible subpath, as typically done by abstract refinement algorithms (lines 10–15). We call the vertices obtained from such splittings that belong to the same property a *property refinement group* (subsets of G; the function *getGroup*(G, v) returns the subset containing v).
2. The second part of the proof of Lem. 1 gives us an $\mathcal{O}(n^2)$ algorithm *GetCoveringPath* for finding a (non-minimal) covering path from I to F in the transitive closure of a directed graph (see Alg. 5), taking into account that a covering path needs to cover only one vertex for each property refinement group (called in line 16 of Alg. 4).
3. A solution π obtained that way might be far from optimal, so we exploit the TSP solver to give us a better solution π'. However, the refined graph does not encode the desired TSP problem because it is sufficient to cover only one vertex for each property refinement group. Hence, given a path π, we transform the graph by collapsing each property refinement group with respect to π as illustrated by Fig. 6 (lines 18–26 of Alg. 4). The obtained graph is handed over to the TSP solver (line 27). Note that the transformations do not preserve optimality, because, e.g. in Fig. 6, the edge (φ_1, φ_2) would cover φ in a concrete path but not in the transformed, refined abstract graph.

Algorithm 4: *GetChain* with abstraction refinement

Input: weighted, directed graph (V, E, W), path π, transition relation T
Output: test case chain $\chi = \langle i_0, \ldots, i_N \rangle$

```
1  G ← {{v} | v ∈ V} //property refinement groups
2  while true do
3      (feasible, χ, failed_path) ← CheckPath(π, T, W)
4      if feasible then return χ
5      (succeeded, W′, failed_path) ← RepairPath(failed_path, T, W)
6      if succeeded then
7          (_, χ, _) ← CheckPath(π, T, W′)
8          return χ
9      (φ′, φ, φ″) = failed_path
10     V ← V ∪ {v_new}
11     getGroup(G, φ) ← getGroup(G, φ) ∪ {v_new}
12     E ← E ∪ {(φ′, v_new)}
13     W(φ′, v_new) ← W(φ′, φ″)
14     E ← E \ {(φ′, φ)}
15     E ← E ∪ {(v_new, v) | (φ, v) ∈ E \ {φ, φ″)}}
16     π ← GetCoveringPath(V, E, G)
17     if π = ⟨⟩ then abort "no chain found for given bound K"
18     foreach v̄ ∈ π do
19         foreach v ∈ getGroup(G, v̄) do
20             if v ≠ v̄ then
21                 E′ = {(v′, v″) | (v′, v) ∈ E ∧ (v, v″) ∈ E ∧ (v′, v″) ∉ E}
22                 foreach (v′, v″) ∈ E′ do
23                     E ← E ∪ {(v′, v″)}
24                     W(v′, v″) ← W(v′, v) + W(v, v″)
25                     E ← E \ E′
26                     V ← V \ {v}
27     π ← GetShortestPath(V, E, W)
```

4. We try to compute a concrete test case chain for the covering path (lines 3–8). If this fails, we iterate the refinement process.

In each iteration (line 2) of the abstraction refinement algorithm, a node in the graph is split such that a concrete spurious transition is removed from the abstraction, i.e. the transition system structure of the program inside the property assumptions is made explicit in the abstraction. Provided the existence of a test case chain, since there is only a finite number of transitions, the abstraction refinement will eventually terminate, and a covering path will be found that can be concretised to a test case chain.

Example 4 (Abstraction refinement). Assume, as in the previous example, that the right state in φ_1 in Fig. 4 is not reachable from the left state. Then the abstraction refinement will split φ_1 into two vertices. Suppose that *GetCover-*

Algorithm 5: *GetCoveringPath*

Input: transitive closure of directed graph (V, E), property refinement groups G
Output: covering path π

```
1 v ← chooseFrom(V); V ← V \ getGroup(G, v); π ← ⟨v⟩
2 while V ≠ ∅ do
3     v ← chooseFrom(V); V ← V \ getGroup(G, v); v' ← lastElement(π)
4     if (v', v) ∈ E then π ← append(π, v)
5     else if (v, v') ∈ E then
6         while (v', v) ∉ E do v' ← previousElement(π, v')
7         π ← insertAfter(π, v, v')
8     else return ⟨⟩ //no path found
9 return π
```

ingPath (Alg. 5) returns the covering path $\pi = \langle I, \varphi_2, \varphi_1, \varphi_2, F\rangle$.[2] Then collapsing the two nodes belonging to φ_1 w.r.t. π will remove the edge from I to φ_1. The TSP solver will optimise π and find the shorter path $\langle I, \varphi_2, \varphi_1, F\rangle$.

Path constraints.. The fundamental problem about a failed path is that it represents information about at least two edges that we cannot encode as an equivalent TSP. We would need a TSP solver that can deal with side conditions like the following: the solution must not contain vertices v_1, v_2, v_3 in this particular order for any infeasible subpath $\langle v_1, v_2, v_3\rangle$ in *failed_path*. Similar difficulties arise concerning *minimality*: here, we would have to add "path weights" that penalise a solution if it contains a certain path. Since our experimental results (§6) suggest that the bottleneck of the approach lies rather in solving reachability queries than TSPs, we can opt for using answer set programming (ASP) solvers (e.g. [11]), which are far less efficient in solving TSPs, but they allow us to specify arbitrary side conditions.

Example 5 (Path constraints). Consider the graph in Fig. 4. We can encode the TSP problem in ASP as follows (cf. [11]):

```
V(I,phi1,phi2,F).

E(I,phi1). weight(I,phi1,0).
E(I,phi2). weight(I,phi2,2).
E(phi1,phi2). weight(phi1,phi2,1).
E(phi1,F). weight(phi1,F,2).
E(phi2,phi1). weight(phi2,phi1,2).
E(phi2,F). weight(phi2,F,2).

{ cycle(X,Y) : E(X,Y) } I :- V(X).
{ cycle(X,Y) : E(X,Y) } I :- V(Y).

reached(Y) :- cycle(I,Y).
reached(Y) :- cycle(X,Y), reached(X).
```

[2] It will actually return the better result for this particular example.

```
:- V(Y), not reached(Y).

#minimize [ cycle(X,Y) : weight(X,Y,C) = C ].
```

Assume, again, that the right state in φ_1 in Fig. 4 is not reachable from the left state so that we obtain $\mathit{failed\_path} = \langle I, \varphi_1, \varphi_2 \rangle$. Then we can exclude $\mathit{failed\_path}$ by adding

```
twopath(X,Y,Z) :- cycle(X,Y), cycle(Y,Z).
-twopath(I,phi1,phi2).
```

to the ASP problem. The ASP solver will return the shortest covering path that does not contain $\mathit{failed\_path}$, i.e. $\langle I, \varphi_2, \varphi_1, F \rangle$.

4.3 Multiple Chains

We can relax our problem to systems that do not admit single chains. Those systems still have to satisfy conditions (1) and (2) of Lem. 1 in order to guarantee the existence of multiple covering chains.

We can detect that a system does not admit a single chain if

- the N-reachability property graph has no chain (where N is the reachability diameter of the system), or
- the chain repair or abstraction refinement process fails.

Algorithm 6: Multiple chains: Partitioning the vertex (property) set such that each partition element admits a single chain

Input: directed graph (V, E)
Output: partition S of V

$R =$ set of pairs $(v_i, v_j) \in V$ that do not satisfy condition (3) of Lem. 1.
$S \leftarrow \emptyset$, $Q \leftarrow V$
for all $(v_i, v_j) \in R$ **do**
  $Q \leftarrow Q \setminus \{v_i, v_j\}$
  if $S = \emptyset$ **then** $S \leftarrow \{(\{v_i\}, \{v_j\}), (\{v_j\}, \{v_i\})\}$
  else
    for all $P = (P^+, P^-) \in S$ **do**
      if $v_i \in P^+ \wedge v_j \in P^+$ **then** $S \leftarrow S \setminus P$
      else if $v_i \in P^- \wedge v_j \notin P^- \wedge v_j \notin P^+$ **then** $P \leftarrow (P^+ \cup \{v_j\}, P^-)$
      else if $v_i \notin P^- \wedge v_j \notin P^- \wedge v_j \in P^+$ **then** $P \leftarrow (P^+, P^- \cup \{v_i\})$
      else if $v_j \in P^- \wedge v_i \notin P^- \wedge v_i \notin P^+$ **then** $P \leftarrow (P^+ \cup \{v_i\}, P^-)$
      else if $v_j \notin P^- \wedge v_i \notin P^- \wedge v_i \in P^+$ **then** $P \leftarrow (P^+, P^- \cup \{v_j\})$
      else if $v_i \notin (P^+ \cup P^-) \wedge v_j \notin (P^+ \cup P^-)$ **then**
      $S \leftarrow S \cup (P^+ \cup \{v_i\}, P^- \cup \{v_j\})$; $P \leftarrow (P^+ \cup \{v_j\}, P^- \cup \{v_i\})$
$S^+ = \mathit{MinCover}(V, \{P^+ \mid (P^+, \_) \in S\})$
choose $P^+ \in S^+$: $P^+ \leftarrow P^+ \cup Q$
for all $P^+ \in S^+$ **do** $P^+ \leftarrow P^+ \cup \{I, F\}$
return S^+

Algorithm 7: *CheckPath*

Input: path π, transition relation T, weights W
Output: whether π is *feasible*, *inputs* associated to π if feasible, *failed_path* $\subseteq \pi$ if infeasible

```
1 inputs ← ⟨⟩
2 failed_path ← ⟨⟩
3 (feasible, assignment, unsat_core) = SAT(BuildPath(π, T, W))
4 if feasible then
5     let (s_0, i_0, s_1, i_1, ..., s_K, i_K) = assignment
6     inputs ← ⟨i_0, ..., i_N⟩
7 else
8     failed_path ← getFailedPath(unsat_core, π)
9 return (feasible, inputs, failed_path)
```

We use Lem. 1 to devise an algorithm for computing a partition $\{P_1, \ldots, P_n\}$ of P (see Alg. 6) and apply Alg. 1 for each P_i. If the chain repairing fails for a P_i, we compute a partition for the refined property graph. Finding the smallest partition is equivalent to the problem of finding a vertex colouring with minimal chromatic number (NP-hard). In Alg. 6, the set S contains pairs of sets (P^+, P^-). P^+ contains the vertices that will form an equivalence class. P^- keeps track of the vertices that are not allowed to be added to P^+. Lines 3 to 13 compute all subsets of V that are consistent with condition (3) of Lem. 1 (F). Line 14 removes the redundant subsets (minimal set cover) and, finally, in line 15 and 16, the remaining vertices Q are added to some element of the partition, and I and F are added to all partitions.

5 Test-Case Generation with Bounded Model Checking

The previous sections abstract from the actual backend implementation of the functions *GetKreachEdges*, *CheckPath*, and *RepairPath*. In this work, we use bounded model checking to provide an efficient implementation. Alternative instantiations could be based on symbolic execution, for example.

BMC-based test case generation. Bounded model checking (BMC) [12] can be used to check the existence of a path $\pi = \langle s_0, s_1, \ldots, s_K \rangle$ of increasing length K from ϕ to ϕ'. This check is performed by deciding satisfiability of the following formula using a SAT solver:

$$\phi(s_0) \wedge \bigwedge_{1 \leq k \leq K} T(s_{k-1}, i_{k-1}, s_k) \wedge \phi'(s_K) \tag{1}$$

If the SAT solver returns the answer *satisfiable*, it also provides a satisfying assignment $(s_0, i_0, s_1, i_1, \ldots, s_{K-1}, i_{K-1}, s_K)$. The satisfying assignment represents one possible path $\pi = \langle s_0, s_1, \ldots, s_K \rangle$ from ϕ to ϕ' and identifies the corresponding input sequence $\langle i_0, \ldots, i_{K-1} \rangle$. Hence, a test case $\langle i_0, \ldots, i_{K-1} \rangle$ covering a

Algorithm 8: *BuildPath*

Input: path π, transition relation T, weights W
Output: path formula Φ

```
return BuildPathRec(π, 0, true)
function BuildPathRec(π, k, Φ)
    if π = ⟨(φ, _)⟩ then
        return Φ ∧ φ(s_k)
    else
        let (v, π_tail) = π
        let (v', _) = π_tail
        let k_end = k + W(v, v')
        let (φ, ψ) = v
        return Φ ∧ φ(s_k, i_k) ∧ ψ(s_{k+1}) ∧ ⋀_{k+1≤j≤k_end} T(s_{j-1}, s_j) ∧
        BuildPathRec(π_tail, k_end, Φ)
```

Algorithm 9: *GetKreachEdges*

Input: weighted, directed graph (V, E, W), transition relation T, edges to be considered E_S, number of steps K
Output: K-reach edges $E_K \subseteq E_S$

```
from_to ← E_S
E_K ← ∅
(sat, assignment) ← checkKreach(from_to, T, K)
while sat do
    let (s_0, i_0, s_1, i_1, ..., s_K, i_K) = assignment
    for all v, v' ∈ V : (φ, ψ) = v, (φ', _) = v' : φ(s_0, i_0) ∧ ψ(s_1) ∧ φ'(s_K) do
        E_K ← E_K ∪ {(v, v')}
        from_to ← from_to \ {(v, v')}
    (sat, assignment, _) ← checkKreach(from_to, T, K)
return E_K
```

property with assumption $\varphi(s, i)$ can be generated by checking satisfiability of a path from I to φ.

Instantiation. For implementing Alg. 1 with chain repair (Alg. 3) we have to provide the functions *CheckPath*, *GetKreachEdges*, and *RepairPath*.

We consider a SAT solver to be a function $SAT : \phi \mapsto (sat, assignment, unsat\_core)$ where *assignment* contains a satisfying assignment if ϕ is *sat* and otherwise *unsat_core* is a minimal formula such that $\phi \Rightarrow unsat\_core$ and $\neg unsat\_core \Rightarrow \neg\phi$.[3]

Then *CheckPath* is defined as in Alg. 7 where *BuildPath* (Alg. 8) constructs the BMC formula for a given path, and *getFailedPath* converts an *unsat_core* into a path (which is SAT solver-specific).

[3] There are alternatives to unsatisfiability cores, e.g., the final conflict feature of MINISAT [13].

Algorithm 10: *RepairPath* by concrete chaining

Input: *failed_path*, transition relation T, weights W, reachability bound K
Output: updated weights W

```
1  σ ← FirstElement(failed_path)
2  for all e = (φ_j, φ_{j+1}) ∈ failed_path do
3      feasible ← false
4      while ¬feasible do
5          (sat, assignment, _) ← CheckPath(⟨σ, φ_{j+1}⟩, T, W)
6          if ¬feasible then W(e) ← W(e) + 1
7          else
8              let ⟨s_0, ...⟩ = assignment
9              σ ← s_0
10         if W(e) > K then return (false, W, ⟨φ_{j−1}, φ_j, φ_{j+1}⟩)
11 return (true, W, ⟨⟩)
```

GetKreachEdges is given as Alg. 9, where the function $checkKreach(\pi, T, K)$ that is used for enumerating K-reachability edges is implemented by checking satisfiability of the following formula:

$$\left(\bigvee_{(\varphi,\varphi') \in E_{target}} \varphi(s_0, i_0) \wedge \varphi'(s_K) \right) \wedge \bigwedge_{1 \leq k \leq K} T(s_{k-1}, i_{k-1}, s_k) \qquad (2)$$

We iteratively check this formula using incremental SAT solving, "removing" the respective terms from the formula each time a solution satisfies (φ, φ'), until the formula becomes unsatisfiable. In addition to assumptions on the inputs, T must also contain a state invariant, obtained, e.g. with a static analyser. This is necessary because, otherwise, the state satisfying φ in Eq. 2 might be unreachable from an initial state.

For the chain repair *RepairPath*, the most efficient method that we tested was to sequentially find a feasible weight for each of the edges in *failed_path*, starting the check for an edge $(\varphi_j, \varphi_{j+1})$ from a concrete state in φ_j obtained from the successful check of the previous edge $(\varphi_{j-1}, \varphi_j)$. This algorithm is listed in Alg. 10.

6 Experimental Evaluation

Implementation. For our experiments we have set up a tool chain (Fig. 7) that generates C code from Simulink models using the Gene-Auto[4] code generator. Our test case chain generator ChainCover[5] itself is built upon the infrastruc-

[4] `http://geneauto.gforge.enseeiht.fr`, version 2.4.9
[5] `http://www.cprover.org/chaincover/`, version 0.1

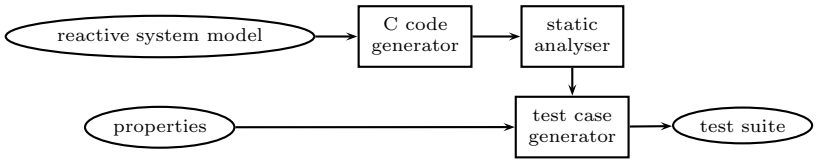

Fig. 7. Tool chain

ture provided by CBMC[6] [14] with MINISAT[7] as a SAT backend, the LKH TSP solver[8] [15], and the CLINGO ASP solver[9] [11].

The properties are written in C using the `assert` and `__CPROVER_assume` macros. For instance, property p_1 in our example is stated as follows:

```
void p_1(t_input* i, t_state* s) {
  __CPROVER_assume(s->mode==ON && s->speed==1 && i->dec);
  compute(i,s);
  assert(s->speed==1);
}
```

Assumptions on the inputs and the state invariant obtained from the static analysis are written as C code in a similar way.

Benchmarks. Our experiments are based on SIMULINK models, mainly from automotive industry. For some benchmarks, we had the SIMULINK models or at least the generated C code available; for others we only had screenshots from the SIMULINK models, which we had to re-engineer ourselves. Our benchmarks are a simple *cruise* control model [2], a *window* controller[10], a car *alarm* system[11], an elevator model [16], and a model of a *robot arm* that can be controlled with a joystick. We generated test case chains for these examples for specifications of different size and granularity. The benchmark characteristics are listed in Table 1. Apart from *Cruise 1* all specifications have properties with multi-state assumptions, thus, the obtained test case chains are not minimal in general. All our benchmarks are (almost) strongly connected (some have an initial transition after which the system is strongly connected), hence, they did not require abstraction refinement.

Comparison. We have compared our tool CHAINCOVER (using LKH) with

- FSHELL[12] [17,18], an efficient test generator with test suite minimisation,
- an in-house, simple *random* case generator with test suite minimisation, and
- KLEE[13] [19], a test case generator based on symbolic execution.

[6] `http://www.cprover.org/cbmc/`, version 4.4
[7] `http://minisat.se`, version 2.2.0
[8] `http://www.akira.ruc.dk/~keld/research/LKH/`, version 2.0.2
[9] `http://potassco.sourceforge.net/`, version 3.0.5
[10] `http://www.mathworks.co.uk/products/simulink/examples.html`
[11] `http://www.mogentes.eu/public/deliverables/MOGENTES_3-15_1.0r_D3.4b_TestTheories-final_main.pdf`
[12] `http://forsyte.at/software/fshell/`
[13] `http://klee.llvm.org/`

	size			ChainCover			FShell			random			KLEE		
benchmark	s	i	P	tcs	len	time	tcs	len	time	tcs	len	time	tcs	len	time
Cruise 1	3b	3b	4	1	9	0.77	3	18	3.67	2.8	24.6	0.54	3	27	46.5
Cruise 2	3b	3b	9	1	10	0.71	4	20	3.56	2.4	21.2	0.07	3	30	17.7
Window 1	3b+1i	5b	8	1	24	14.1	4	32	19.0	1.8	40.4	58.9	3	72	155
Window 2	3b+1i	5b	16	1	45	24.9	7	56	28.3	2.0	86.8	18.7	5	225	242
Alarm 1	4b+1i	2b	5	1	26	7.51	1	27	509	80% cov.		t/o	60% cov.		t/o
Alarm 2	4b+1i	2b	16	1	71	33.5	3	81	690	94% cov.		t/o	63% cov.		t/o
Elevator 1	6b	3b	4	1	8	22.9	2	15	115	2.2	10.4	0.85	2	16	24.4
Elevator 2	6b	3b	10	1	32	97.3	5	54	789	2.6	49.0	65.8	70% cov.		t/o
Elevator 3	6b	3b	19	1	48	458	6	54	838	4.0	149	18.0	53% cov.		t/o
Robotarm 1	4b+2f	3b	4	1	25	185	2	22	362	2.4	49.0	0.07	2	40	10.9
Robotarm 2	4b+2f	3b	10	1	47	113	2	33	532	3.8	72.2	0.21	80% cov.		t/o
Robotarm 3	4b+2f	3b	18	1	84	427	5	55	731	3.2	160	0.62	67% cov.		t/o

Table 1. Experimental results: The table lists the number of test cases/chains (tcs), the accumulated length of the test case chains (len), and the time (in seconds) taken for test case generation. Size indicates the size of the program in the number of (minimally encoded) Boolean (b), integer (i) and floating point (f) variables and (minimally encoded) Boolean (b) inputs. "P" is the number of properties in the specification. If the tool timed out ("t/o") after 1 hour the achieved coverage ("cov") is given.

In order to make results comparable, we have chosen F to be equivalent to I (or the state after the initial transition). Hence, test cases generated by FShell, random, and Klee can be concatenated (disregarding the initial transition) to get a single test case chain.

Like our tool, FShell is based on bounded model checking. FShell takes a coverage specification in form of a query as input. It computes test cases that start in I, cover one or more properties $p_1, \ldots, p_n$ and terminate in F when given the query: `cover (@CALL(p_1) | ... | @CALL(p_n)) -> @CALL(final)`. In the best case, FShell returns a single test case, i.e. a test chain. We have run FShell with increasing unwinding bounds K until all properties were covered.

For random testing and Klee, we coded the requirement to finish a test case in F with the help of flags in the test harness. Then we stopped the tools as soon as full coverage was achieved and selected the test cases achieving full coverage while minimising the length of the input sequence using an in-house, weighted-minimal-cover-based test suite minimiser. For random testing we averaged the results over five runs. Unlike ChainCover and FShell, which start test chain computation without prior knowledge of how many steps are needed to produce a test case, we had to provide random testing and Klee with this information. The reason is that the decision when a certain number of steps will not yield a test case can only be taken after reaching a timeout for random testing. Similarly, Klee may take hours to terminate. Consequently, the results for random testing and Klee are not fully comparable to those of the other tools.

Results. Experimental results obtained are shown in Table 1 and Fig. 8.

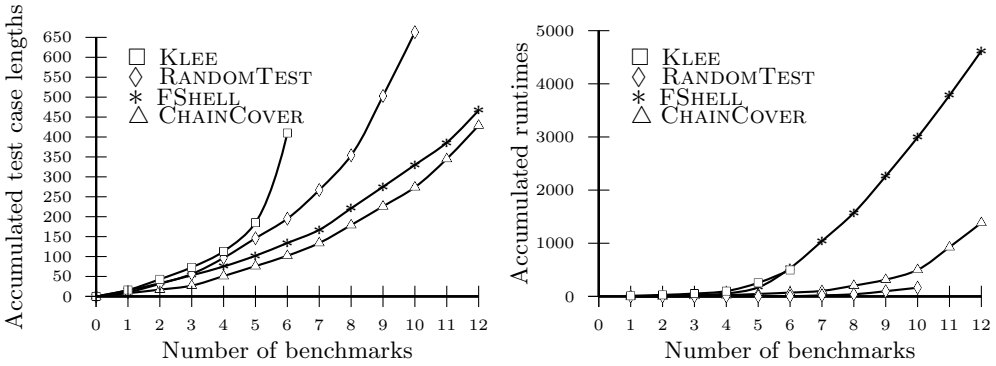

Fig. 8. Experimental results: accumulative graph of test case lengths on the left-hand side, accumulated runtimes on the right-hand side.

- Our tool ChainCover usually succeeds in finding shorter test case chains than the other tools. It is also in general faster. ChainCover spends more than 99% of its runtime with BMC. The time for solving the ATSP problem is neglible for the number of properties we have in the specifications. The runtime ratio for generating the property K-reachability graph ($\mathcal{O}(Kn^2)$ BMC queries for n properties) versus finding and repairing a chain ($\mathcal{O}(Kn)$ BMC queries) varies between 7:92 and 75:24.
- FShell comes closest to ChainCover with respect to test case chain length, and finds shorter chains on the robot arm example. However, FShell takes much longer: the computational cost depends on the number of unwindings and the size of the program and less on the number of properties.
- Random testing yields very good results on some (small) specifications and sometimes even finds chains that are as short as those generated by ChainCover. However, the results vary and heavily depend on the program and the specification: in some cases, e.g. *Robotarm*, full coverage is achieved in fractions of a second; in other cases, full coverage could not be obtained before reaching the timeout of one hour and generating millions of test cases.
- Klee found test case chains on a few of the benchmarks in very short time, but did not achieve full coverage within an hour on half of the benchmarks, which suggests that exhaustive exploration is not suitable for our problem.

7 Related Work

Test case generation with model checkers came up in the mid-90s and has attracted continuous research interest since then, especially due to the enormous progress in SAT solver performance. There is a vast literature on this topic, surveyed in [20], for example. The FShell tool [18,17] we have compared with was developed with the motivation of enabling the flexible specification of the desired coverage.

Reactive system testing. There are many approaches to reactive system testing: While random testing [21] is still commonly used, approaches have

been developed that combine random testing with *symbolic and concrete execution* (DART [22], CUTE [23], KLEE [19]) to guide exhaustive path enumeration. *Scenario-based testing* employ test specifications to guide test case generation towards a particular functionality (e.g., LUTESS [24], LURETTE [25], LUTIN [26]). These methods restrict the input space using static analysis and apply (non-uniform) random test case generation. *Model-based testing* (see [27,28] for surveys on this topic) considers specification models based on labelled transition systems. For instance, extended finite state machines (EFSM) [29,30,31] are commonly used in communication protocol testing to provide exhaustive test case generation for conformance testing. Available tools include, e.g., TGV [32] and TORX [33].

Minimal checking sequences and test optimisation. In the model-based testing domain, the problem of finding minimal checking sequences has been studied in *conformance testing* [3,4,1,5,6], which amounts to checking whether each state and transition in a given EFSM specification is correctly implemented. First, a minimal checking path is computed, which might be infeasible due to the operations on the data variables. Subsequently, random test case generation is applied to discover such a path, which might fail again. Duale and Uyar [34] propose an algorithm for finding a feasible transition path, but it requres guards and assignments in the models to be linear. Another approach is to use genetic algorithms [3,35] to find a feasible path of minimised length. Also in our setting, the use of genetic algorithms in order to find minimised instead of minimal solutions is an option to consider. SAT solvers have also been used to compute (non-minimal) checking sequences in FSM models [36,37]. Our method does not impose restrictions on guards and assignments and implicitly handles low-level issues such as overflows and the semantics of floating-point arithmetic in finding feasible test cases. The fact that minimal paths on the abstraction might not be feasible in the concrete program does not arise due to limited reasoning about data variables, but due to the multi-state nature of the properties we are trying to cover.

Closest to our work is recent work [38] on generating test chains for EFSM models with timers. They use SMT solvers to find a path to the nearest test goal and symbolic execution to constrain the search space. If no test goal is reachable they backtrack to continue the search from an earlier state in the test chain. Their approach represents a greedy heuristics and thus makes minimality considerations difficult. Our method can handle timing information if it is explicitly expressed as counters in the program.

Petrenko et al [39] propose a method for test optimisation for EFSM models with timers. They use an ATSP solver to find an optimal ordering of a given set of test cases and an SMT solver to determine paths connecting them. The problem they tackle is easier than ours because they do not generate test cases, but just try to chain a given set of test cases in an optimal order. Additionally, they take into account overlappings of test cases during optimisation.

In contrast to all these works, our approach starts from a partial specification given by a set of properties, usually formalised from high-level requirements. The

K-reachability graph abstraction can be viewed as the generation of a model from a partial specification and automated annotation of model transitions with timing information in terms of the minimal number of steps required.

8 Summary and Prospects

We have presented a novel approach to discovering a minimal test case chain, i.e., a single test case that covers a given set of test goals in a minimal number of execution steps. Our approach combines reachability analysis to build an abstraction, TSP-based optimisation and heuristics to find a concrete solution in case we cannot guarantee minimality. The test goals might also be generated from an EFSM specification or from code coverage criteria like MC/DC. This flexibility is a distinguishing feature of our approach that makes it equally applicable to model-based and structural coverage-based testing. In our experimental evaluation, we have shown that our tool ChainCover outperforms state-of-the-art test suite generators. Moreover, our approach is not restricted to C code generated from Simulink—it can be applied to any reactive system language. For instance, we could also consider consider Verilog, or the application to HW/SW-co-verification combing Verilog and C code.

Prospects. Deep loops pose a problem for BMC-based methods. For instance, we had to reduce size of loop bound constants in the car *alarm* system benchmark to make it tractable for comparison. Acceleration methods, e.g. [40], are expected to remedy many such situations, especially those involving counters.

Moreover, the property K-reachability graph generation lends itself to parallellisation. This is expected to give a further boost to the capacity of our tool.

Test case chains are intended to demonstrate conformance in late stages of the development cycle, especially in acceptance tests when the system can be assumed stable. It is an interesting question in how far they can be used in earlier phases: The test case chains computed by our method are able to continue to the subsequent test goals even if a test fails, as long as the implementation has not changed too much; otherwise the test chain has to be recomputed. In this case, it would be desirable to incrementally adapt the test case chain after bug fixes and code changes.

Acknowledgements. We thank Cristian Cadar for his advice regarding the comparison with Klee, and the anonymous reviewers for their invaluable comments.

References

1. Hierons, R., Ural, H.: Generating a checking sequence with a minimum number of reset transitions. ASE **17** (2010) 217–250
2. Robert Bosch GmbH: Bosch Automotive Handbook. Bentley (2007)
3. Nuñez, A., Merayo, M., Hierons, R., Núñez, M.: Using genetic algorithms to generate test sequences for complex timed systems. Soft Computing **17** (2013) 301–315

4. Petrenko, A., da Silva Simão, A., Yevtushenko, N.: Generating checking sequences for nondeterministic finite state machines. In: ICST. (2012) 310–319
5. Hierons, R., Ural, H.: Optimizing the length of checking sequences. Trans. on Computers **55** (2006) 618–629
6. Hierons, R.: Using a minimal number of resets when testing from a finite state machine. Inf. Proc. Letters **90** (2004) 287 – 292
7. Boyd, S., Ural, H.: On the complexity of generating optimal test sequences. Trans. Softw. Eng. **17** (1991) 976–978
8. Floyd, R.: Algorithm 97: Shortest path. Communications of the ACM **5** (1962) 345
9. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. ENTCS **66** (2002) 160–177
10. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: VMCAI. Volume 2575 of LNCS. (2003) 298–309
11. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: The Potsdam answer set solving collection. AI Communications **24** (2011) 107–124
12. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19** (2001) 7–34
13. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Formal Methods in Computer-Aided Design. (2010) 181–188
14. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Volume 2988 of LNCS. (2004) 168–176
15. Helsgaun, K.: An effective implementation of the Lin-Kernighan traveling salesman heuristic. European J. of Operational Research **126** (2000) 106–130
16. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: TAP. Volume 6706 of LNCS. (2011) 134–151
17. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: CAV. Volume 5123 of LNCS. (2008) 209–213
18. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: VMCAI. Volume 5403 of LNCS. (2009) 151–166
19. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. (2008) 209–224
20. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Software Testing, Verification & Reliability **19** (2009) 215–261
21. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. Trans. Softw. Eng. **10** (1984) 438–444
22. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. (2005) 213–223
23. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: CAV. Volume 4144 of LNCS. (2006) 419–423
24. du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: A specification-driven testing environment for synchronous software. In: ICSE. (1999) 267–276
25. Jahier, E., Raymond, P., Baufreton, P.: Case studies with Lurette V2. STTT **8** (2006) 517–530
26. Raymond, P., Roux, Y., Jahier, E.: Lutin: A language for specifying and executing reactive scenarios. EURASIP J. on Embedded Systems (2008)

27. Petrenko, A., da Silva Simão, A., Maldonado, J.C.: Model-based testing of software and systems: recent advances and challenges. STTT **14** (2012) 383–386
28. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines – a survey. Proc. IEEE **84** (1996) 1090–1123
29. Lee, D., Yannakakis, M.: Optimization problems from feature testing of communication protocols. In: Int. Conf. on Netw. Protocols. (1996) 66
30. Ural, H., Yang, B.: A test sequence selection method for protocol testing. IEEE Trans. on Comm. **39** (1991) 514–523
31. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. Trans. Softw. Eng. **30** (2004) 29–42
32. Jard, C., Jéron, T.: TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT **7** (2005) 297–315
33. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. Volume 4949 of LNCS. (2008) 1–38
34. Duale, A., Uyar, M.Ü.: A method enabling feasible conformance test sequence generation for EFSM models. IEEE Trans. Computers **53** (2004) 614–627
35. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: ICST. (2009) 230–239
36. Jourdan, G.V., Ural, H., Yenigün, H., Zhu, D.: Using a SAT solver to generate checking sequences. In: Int. Sym. on Comp. and Inf. Sciences. (2009) 549–554
37. Mori, T., Otsuka, H., Funabiki, N., Nakata, A., Higashino, T.: A test sequence generation method for communication protocols using the SAT algorithm. System and Computers in Japan **34** (2003) 20–29
38. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: NASA Formal Methods. Volume 6617 of LNCS. (2011) 298–312
39. Petrenko, A., Dury, A., Ramesh, S., Mohalik, S.: A method and tool for test optimization for automotive controllers. In: Software Testing, Verification and Validation Workshops. (2013) 198–207
40. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. In: CAV. Volume 8044 of LNCS. (2013) 381–396