Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 320 (2016) 35–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Test Data Generation for Cyclic Executives with CBMC and Frama-C: A Case Study

Omer Nguena Timo [1,2] Guillaume Langelier [1,3]

Centre de recherche informatique de Montréal
Montréal, Canada

Abstract

Testers of cyclic executive systems are required to make considerable and repetitive efforts to determine input sequences (tests) for leading a system from a start state to a target state. This task is time-consuming and it might lead testers to produce fewer tests than required; which affects negatively the quality of systems and can reduce time-to-market. We propose an automated test generation approach which integrates two renowned tools: the code slicing plugin of Frama-C and the bounded model-checker CBMC. We also suggest several code metrics to better assess the complexity of code and the effect of code slicing on test derivation with CBMC. The proposed approach has been tested on industrial case studies and significantly reduces test generation effort and computation time on two automotive controllers. The proposed approach alleviates the workload of testers so they are able to focus on producing more tests; thus, increasing the quality of systems.

Keywords: Test generation, cyclic executive, C Code, automotive controllers, experimentation, CBMC, Frama-C

1 Introduction

Controller software components in the automotive domain implement real-time reactive systems with C Code which are often automatically generated from Simulink/Stateflow (SL/SF) [27] models that are based on the cyclic executive model [2,24]. The cyclic executive model decomposes a component into cooperating non-preemptive periodic tasks which operate in an execution loop. Every iteration step of the executive loop consists of three phases: the reading of inputs from the environment, the computation of outputs which are also the state of the controller, and the transmission of the outputs to the environment.

[1] This work is supported by NECSIS (Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems) of NSERC, Canada. We would like to acknowledge Alexandre Petrenko, lead researcher of MODL team at CRIM, for his comments and revision on early versions of this paper.

[2] Email: omer.nguena-timo@crim.ca

[3] Email: guillaume.langelier@crim.ca

http://dx.doi.org/10.1016/j.entcs.2016.01.004

1571-0661/© 2016 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Engineers dealing with controller software components need to generate tests to ensure their quality and safety. Certain tests may aim at validating the activation of interesting control commands that can only be executed at certain states. So, a test is represented by an input sequence and a target state. It can be difficult for an expert to determine an input sequence leading to a given target in a complex component because of numerous code paths and variable interactions. Moreover, this task is repetitive, time-consuming and error-prone. To automate this process and make it more efficient, engineers need to be supported by tools.

The test generation problem we address is formulated as follows: given two sets of states *Start* and *Target* of a cyclic executive implementation of a controller software component in the C language, generate an input sequence leading the component from one state in *Start* to some state in *Target*. State sets are defined by constraints on output variables.

In order to solve the test generation problem we adapt the approach based on bounded model checking, solvers and slicing in a framework that integrates two off-the-shelf renowned tools: Frama-C [16] and CBMC [11,23]. The adaptation includes: (a) a specialised slicing engine for cyclic executives which uses the slicing plugin of Frama-C and (b) a test generation engine built on top of CBMC. We present experimental results for two industrial powertrain software components comparing our framework with CBMC alone. Results show that slicing of cyclic executive systems is effective and enhances test generation for complex controllers by reducing computation time. We use several code metrics, including the number of acyclic paths, to evaluate the complexity of code hampering model-checking based test generation engines. We also use these metrics to show the importance of slicing in our approach.

The paper is organized as follows: Section 2 defines the cyclic executive model and Section 3 describes the approach of our framework. Section 4 presents the case studies and experimental results, Section 5 discusses related works and the paper ends with conclusions and future work in Section 6.

2 Cyclic Executive Systems

2.1 Cyclic Executive Systems and Semantics

Cyclic executives [2,24] share a common architecture model for real-time embedded systems. They are divided into sets of non-preemptive periodic tasks. Inputs and outputs are shared by all tasks. A set of tasks executable at the same time is called a frame. Cyclic executive systems read the inputs before executing every frame and send the outputs to the environment after executing every frame.

Formally, a *cyclic executive system* (CES) is a tuple $P = \{I, O, Tsk, period, \Theta, prec\}$ where I is a set of *input variables*, O is a set of *output variables*, $Tsk = \{t_1, t_2, \ldots, t_n\}$ is a set of periodic *tasks*, $period : Tsk \to \mathbb{N}^+$ is a map that assigns a period (rational number) to each task, Θ is a Boolean predicate over the output variables and it defines the initial values of the outputs variables, $prec \subseteq Tsk \times Tsk$ is a strict linear order over the tasks which defines

```
1  int x1, x2, x3, x4, x5;
2  int y1, y2 ;
3
4  void taskA_2p00() {
5     if (y1%2==0) {
6     x2 = x1 + 7 ;
7     x3 = x2 + y1 ;
8     } else {
9        x1 = x3 + 2*x2 ;
10       x5 = x5 + y1 ;
11    }
12 }
13
14 void taskB_3p00() {
15       x2 = x2 - 3 ;
16       x4 =  x5 + y2 ;
17 }
```

Listing 1. task.c

```
1  #include "task.c"
2  int time ;
3  void inputReading() { scanf(y1, y2);}
4  void initOutputs() { x1= 0;  ...}
5  void outputTransmit() {printf(x1,...) ;}
6  void transition() {
7       if (time % 6 == 0){
8         taskA_2p00() ;
9         taskA_3p00() ;
10      } else if (time % 2 == 0) taskA_2p00()
            ;
11       else if (time % 3 == 0) taskA_3p00()
            ;
12      time ++ ;
13 }
14
15 void main() {
16    time = 0 ;
17    initOutputs;
18    while(1) {
19    readInputs() ;
20    transition() ;
21    outputTransmit() ;
22    }
23 }
```

Listing 2. scheduler.c

an execution order between the tasks which are executable at the same time. The order of any two tasks is the same for all frames where they are present.

As a running example, we consider a cyclic executive system specified with two input variables: $y1$ and $y2$, five output variables: $x1$, $x2$, $x3$, $x4$ and $x5$; and two tasks: $taskA\_2p00$ and $taskB\_3p00$ which are executed every 2 and 3 time units, respectively. So, the system have three frames, $F_1 = taskA_2p00.taskB_3p00$, $F_2 = taskA_2p00$ and $F_3 = taskB_3p00$. Frame F_1 is executed at time 0 and then every 6 time units. Frame F_2 (resp. F_3) is executed every even (resp. multiple of 3) time units which is not a multiple of 6. Listing 1 presents the skeleton of C code for these tasks.

Let $Dom(V)$ denote the space of values of variables in set V. A *task* t_k can be represented by a total function $t_k : Dom(O) \times Dom(I) \to Dom(O)$ that computes new outputs from current inputs and past outputs. So, a task is not allowed to change input values. The sequential composition of two tasks t_1 and t_2, denoted t_1t_2, represents the application of t_1 immediately followed by the application of t_2 and $t_1t_2(o, i) = t_2(t_1(o, i), i)$. A *state* of P is a valuation of the output variables. The *execution loop* for P repeatedly executes three actions: reading the inputs, calling a transition function, and transmitting the outputs. The transition function executes a frame based on the current execution time. Formally, a *frame* for P is a composition of tasks $F = t_1t_2 \dots t_m$ such that $(t_k, t_{k+1}) \in prec$ for every $k = 1 \dots m-1$ and for every task $t' \in Tsk$ but not in F it holds that $lcm(\{period(t) | t \in F \cup \{t'\}\}) > lcm(\{period(t) | t \in F\})$, where $lcm(S)$ stands for the least common multiple of elements in set S. The set of frames for P is denoted by $\mathcal{F}(P)$. A *transition function* for P is a function $trans_P : \mathbb{N} \to \mathcal{F}(P) \cup \{\perp\}$ such that $trans_P(0) = Tsk$ and $trans_P(\delta) = F$ with $\delta > 0$ iff δ is a multiple of the periods of the tasks in F; otherwise $trans_P(\delta) = \perp$ where $\perp$ denotes a special frame

with no task such that $\perp(o, i) = o$ for every input valuation i and every state o. We consider a discrete time execution of P which calls the transition function at every time unit. An execution of P from $o_0 \models \Theta$ is a sequence $\sigma = o_0 i_1 o_1 i_2 o_2 \dots o_{n-1} i_n o_n$ such that $o_\delta = trans_P(\delta)(o_{\delta-1}, i_\delta)$ for every $1 \leq \delta \leq n$. The input sequence for σ is the sequence $\lambda(\sigma) = i_1 i_2 \cdots i_n$ and the duration of σ is n. Then, $P(\lambda)$ denotes the execution of P with input sequence λ.

2.2 The Test Generation Problem

Many controllers, e.g., automotive controllers, are cyclic executive systems where input variables are updated with sensors and output variables describe command to actuators and physical devices. Transmitting unexpected values of outputs to the environment may lead to catastrophic situations like, e.g, collision of vehicles. Finding an input sequence that fires an unexpected command during an execution of a cyclic executive system is a repetitive and difficult task. A test target τ for a cyclic executive system P is a propositional logic formula over output variables and every atomic proposition is a comparison between linear arithmetic expressions over the variables and the real numbers. An execution $P(\lambda) = o_0 i_1 o_1 i_2 o_2 \dots o_{n-1} i_n o_n$ *reaches* target τ if τ evaluates to true according to the values of the output variables defined in o_n. The test data generation problem for P and τ amounts to compute, when it exists, an input sequence λ such that $P(\lambda)$ reaches τ.

We consider the following three targets for the running example in Listing 1: $x1 = 30$, $x4 = 8$ and $x1 = 32 \wedge x4 = 7$. Then, the test generation problem consists to generate three sequences of valuations of the input variables $y1$ and $y2$.

2.3 Code for Cyclic Executive Systems

In this research work, we generate input sequences for cyclic executive systems implemented in the C language. The nature of the variables (input or output) and the periods of the tasks are specified with external data files. Full implementations of CES are heavily hardware dependent [1]. Consequently testers are often provided with implementations of tasks only and they should take care of implementing the whole scheduler with functions for reading the inputs, transmitting the outputs and computing new outputs along with the main scheduling loop. Listing 1 and Listing 2 show a full code of the running example.

We recall that C functions are sequences of declaration, conditional, assignment and control-flow statements. Detailing the semantics of C language is out of the scope of this paper. We assume that only one statement appears at a line of code. However, we recall that C programs execute statement by statement. The next executable statement is at the next line of code unless a control-flow statement requests a jump to another line of code. Let L denote the set of program statements of P_{impl}. An execution state of P_{impl} is a tuple $s = (o, e)$ where o is a valuation of the output variables and variable *time* immediately before the execution of the statement $e \in L$. For the sake of simplicity, we omit local variables in execution states. A simple execution of P_{impl} from execution state (o, e) with inputs i is

represented with a tuple $((o,e),i,(o',e'))$ where (o',e') is computed according to the semantics of C language. An execution of P_{impl} from an initial state (o_0,e_0) is a sequence $\rho = (o_0,e_0)i_1(o_1,e_1)i_2 \dots i_n(o_n,e_n)$ where $((o_{k-1},e_{k-1}),i_k,(o_k,e_k))$ are simple executions. An *observable* execution states includes the statement which call the function for transmitting the outputs to the environment. The observed output sequence for ρ, $obs(\rho) = o_{k_1}o_{k_2}\dots o_{k_m}$ is the projection of ρ to the outputs in observable states. The input sequence for ρ is the sequence $\lambda(\rho) = i_{k_1}i_{k_2}\dots i_{k_m}$ and the duration of ρ is m. In the sequence $P_{impl}(\lambda)$ denotes an execution of P_{impl} with the input sequence λ. We say that execution $P_{impl}(\lambda)$ fires target τ iff at least one state in $obs(P_{impl}(\lambda))$ satisfies τ. Finally, we say that P_{impl} is a correct implementation of P iff $obs(P_{impl}(\lambda)) = P(\lambda)$ for every input sequence λ. In the sequel, we generate tests from correct implementations of CES.

3 Test Generation with Frama-C and CBMC

The proposed framework for automatic test generation for cyclic executive systems is based on bounded model-checking and it is implemented with CBMC. At the same time, to enhance test generation with complex code, we apply code slicing with Frama-C before the test generation. Generated tests remain valid on the original code. Figure 1 shows the architecture of the test generation framework we propose.

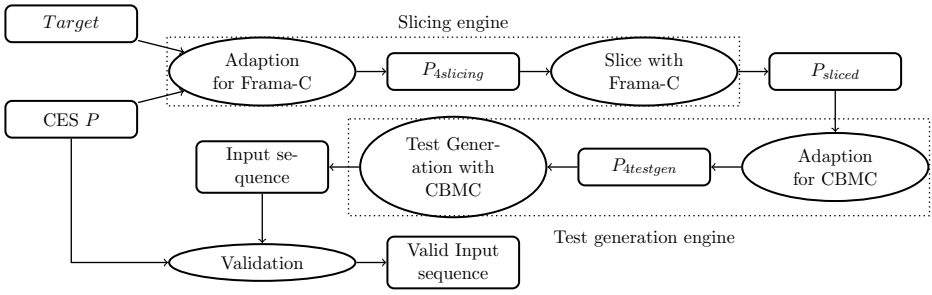


Fig. 1. Test generation framework

The choice of CBMC [11,23] and Frama-C [16] is motivated by their availability, their simplicity, their reputation and ability to handle the whole C language. However, any other tools with similar functionalities can be used in this framework without altering its fundamental principles.

3.1 Bounded Model-Checking based Test Generation with CBMC

Model-checking [12] verifies that systems are satisfying properties and generates a counterexample when systems fail to satisfy them. Counterexamples are system execution traces which include valuation of variables. This paper applies the well-known model-checking based test generation principle [20] and reduces the test data generation problem for cyclic executive systems to a model-checking problem for safety properties. The safety property forbids a system under test to reach a target.

```
1  #include "task.c"
2  ...
3  void inputReading() {
4    y1 = nondet_int() ;
5    ...
6  }
7  ...
8  void main() {
9     time = 0 ;
10    initOutputs();
11    while(1) {
12    readInputs() ;
13    transition() ;
14    assert(!target) ; //
          inserted assertion
15    }
16 }
```

Listing 3. Adaptation for CBMC

```
1  #include "task.c"
2  ...
3  void inputReading() {
4    y1 = Frama_C_Interval(MININT,
         MAXINT) ;
5    ...
6  }
7  ...
8  void main() {
9     time = 0 ;
10    initOutputs();
11    while(time < BOUND) {
12    readInputs() ;
13    transition() ;
14     /*@ slice pragma expr target; */
15    }
16 }
```

Listing 4. Adaptation for Frama-C

However, model-checking engines which perform exhaustive explorations of infinite state spaces of program may not terminate. In particular unbounded loops may trigger the exploration of infinitely many new states, which is not algorithmically solvable. Bounded model-checking [10] explores a restricted finite subset of the state space. The restriction is performed by unrolling the loops finitely many times. In particular, the main execution loop of cyclic executives must be unrolled as many times as the maximum expected execution length for reaching a given target. This bound must be provided to framework. If the target is reachable within this maximum length, the counterexample trace determines the input sequence.

CBMC is a bounded model-checker for ANSI C programs. Using CBMC requires instrumenting the code with the input reading function and the target. The adaptation of the input reading function uses *"nondet_"* primitives defined by CBMC. In order to verify if the target is reached, an assertion on its constraints is checked immediately after each call to the transition function. Listing 3 illustrates the adaptation of the input function and the instrumentation of the target. Once the code instrumentation has been performed, CBMC is executed on the code along with bounds for unbounded loops. These bounds must be determined by test experts who have a good knowledge of the code because CBMC is unable to automatically infer them or choose reasonable values.

To generate an input sequence, CBMC transforms a program and a property under analysis into a SAT/SMT formula which is resolved with SMT [18] or SAT solvers like e.g., MiniSat, boolector, Yices or Z3. The constructed formula is a Boolean combination of atomic propositions built from the single static assignment (SSA) form [17,33] of the program. To build the SSA form, function calls, loops and goto statements are unfolded. Variables are also renamed whenever they are redefined so that every variable is defined only once. The more there are variables, complex operations (division, multiplication), paths in a program, the more complex are the satisfiability checking of the generated formulas and the generation of test data. However, improvements to solvers in recent years allow them to solve very complex problems and to scale up to the analysis of components of industrial embedded systems [11].

3.2 Code Complexity Metrics

Evaluating and reducing the complexity of code that CBMC analyses is substantial for boosting the test generation. Based on the workflow of CBMC, we suggest evaluating code complexity with the following metrics. We refer to [29,37] for detailed presentations of the metrics.

- Number of non-comment source lines (*NCSL*) represents the number of non-declaration statements in a function.
- *NPath* is the number of acyclic execution paths in code. Acyclic execution paths are the longest paths in control flow graph of functions that do not cross a loop's back edge.
- *CCM* is the cyclomatic complexity metric. It is the number of linearly-independent paths [4] through a control flow graph of code.
- *HVM* is the Hasltead Volume Metric. It is the product of the total numbers of operators and operands with the logarithm of the total number of distinct operators and distinct operands.

3.3 Slicing of Cyclic Executive Systems for The Test Generation

We apply code slicing [35] to reduce the complexity of cyclic executive systems under test. The slicing criterion is the preservation at the observation states of the values of the outputs variables which appear in targets. Formally, a slicing criterion for a cyclic executive system under test is specified with a subset $C \subseteq O$ of the output variables. Given an observable state $s = (o, e)$ of P_{impl} and $C \subseteq O$, $s \downarrow C$ narrows o to the values of variables in O, i.e., $s \downarrow C(v) = o(v)$ if $v \in C$; otherwise $s \downarrow C(v)$ is undefined. The projection of $obs(P_{impl}(\lambda)) = s_1 s_2 \dots s_{n-1} s_n$ on a slicing criterion C, $proj_C(obs(P_{impl}(\lambda)))$ is the sequence $(s_1 \downarrow C)(s_2 \downarrow C) \dots (s_{n-1} \downarrow C)(s_n \downarrow C)$.

A program Q is a slice for P_{impl} and a slicing criterion C if Q satisfies the following properties: (1) Q can be obtained from P_{impl} by deleting zero or more statements in P_{imp}. (2) Whenever $P_{imp}(\lambda)$ is terminating for an input sequence λ, Q is also terminating and $proj_C(obs(P_{impl}(\lambda))) = proj_C(obs(Q(\lambda)))$.

In our test generation framework, we propose a slicing engine for cyclic executive systems which use the slicing plugin of Frama-C as a back-end. Frama-C is an open source static C code analyser augmented with many plugins like the program dependency graph and the slicing plugin. The slicing plugin of Frama-C reduces the code by using over-approximations of the values of the variables at every statement along with the control and data dependencies between statements. The dependencies are represented with a collection of dependency graphs; one for each function in the code [19,21]. Statement m is control-dependent on statement n if n is a decision-statement; then the function will execute m for one of the truth-values of the decision, and the function will not execute m for the other truth-value. Statement m is data-dependent statement n if the value of a variable v can be modified

[4] A path in the control flow graph of a function is linearly independent if it introduces at least one new edge that is not included in any other linearly independent paths

by n, m accesses the value of v, and the value of v cannot change between executions from n to m.

Frama-C proceeds as follows [30,21] to slice a function with respect to slicing criterion C. First, it identifies the observable statement where the slicing criterion is evaluated. Then, it deletes the statements which do not occur in any paths of the dependency graphs from the entry statement of the code to the observation statement; this operation proceeds in the graph of every function called between the entry statement of the code and the observation statement. Finally, generate the reduced code by projecting the dependency graph on the remaining statements. The slicing algorithm of Frama-C which uses the value-analysis plug-in of Frama-C is interprocedural and flow, path and context sensitive [35,4], i.e, the order of execution of statements, feasible paths and call-sites matter for code reduction.

This plugin from Frama-C outputs code corresponding to its internal representation. One of the modification performed by Frama-C is to separate decisions in atomic conditions resulting in supplementary conditionals and goto instructions. The result from the execution of the program is exactly the same, but we will see in Section 4 that it can have undesirable effects in some cases.

Our slicing engine for cyclic executives augments the code with the slicing criterion, limits [5] the lengths of the paths that the slicing plugin of frama-C will analyse, and reduces the instrumented code with the slicing plugin of Frama-C. Indeed, if the length of the execution paths is not bounded, the path-sensitive slicing plugin of Frama-C will remove the unbounded loops (e.g the main executive loop) because the execution paths of infinite lengths within these loops are unfeasible; the sliced code will not be correct for the test generation. In our framework, the tester bounds the length of the execution paths; it is recommended to use the same bounds as for the test generation with the bounded model-checker CBMC.

Sliced versions returned by Frama-C for the cyclic executive of our running example are presented in Listing 5 and 6. Assignments on $x4$ and $x5$ are deleted in the sliced version for preserving the value of $x1$ at every cycle of the cyclic executive and they are the only assignments kept in the slice for preserving the value of $x4$. The cyclomatic and the Npath complexity for $taskA\_2p00$ are equal to 2 in all versions of code and the cyclomatic and the Npath complexity for $taskB\_3p00$ are equal to 1 in all versions of code. The values of the metrics was computed with the tool NPath [6].

CBMC is executed for generating input sequences for the three targets assuming maximal execution durations for reaching each of the target. Input sequences computed by CBMC are given in Table 1.

4 Industrial Case Studies

In this section, we apply the proposed approach to generate tests for real automotive controllers and we compare the obtained results with those produced by

[5] Note that options "-slevel" and "-ulevel" cannot be used to this end.

[6] http://www.geonius.com/software/tools/npath.html

```
1  void taskA_2p00_slice_1(void)
2  {
3    if (y1 % 2 == 0) {
4      x2 = x1 + 7;
5      x3 = x2 + y1;
6    }
7    else x1 = x3 + 2 * x2;
8    return;
9  }
10
11 void taskB_3p00_slice_1(void)
12 {
13   x2 -= 3;
14   return;
15 }
```

Listing 5. slice for $x1$

```
1  void taskA_2p00_slice_1(void)
2  {
3    if (! (y1 % 2 == 0)) x5 += y1
       ;
4    return;
5  }
6
7  void taskB_3p00_slice_1(void)
8  {
9    x4 = x5 + y2;
10   return;
11 }
```

Listing 6. slice for $x4$

Target	Max. exec. dur.	Test	Comp. delay (sec.)
$x1 = 30$	7	$(y1)^7 = (-10)(-9)(-1)(-9)(-6)(-9)(1)$	0.27
$x4 = 8$	5	$(y1, y2) = (2, 8)$	0.24
$x1 = 32 \&\& x4 = 7$	7	$(y1, y2)^7 = (-10, -9)(-8, -8)(-3, -9)(-10, 0)(-4, -8)(-9, -10)(5, 5)$	0.27

Table 1
Input sequence derived by CBMC for the running example

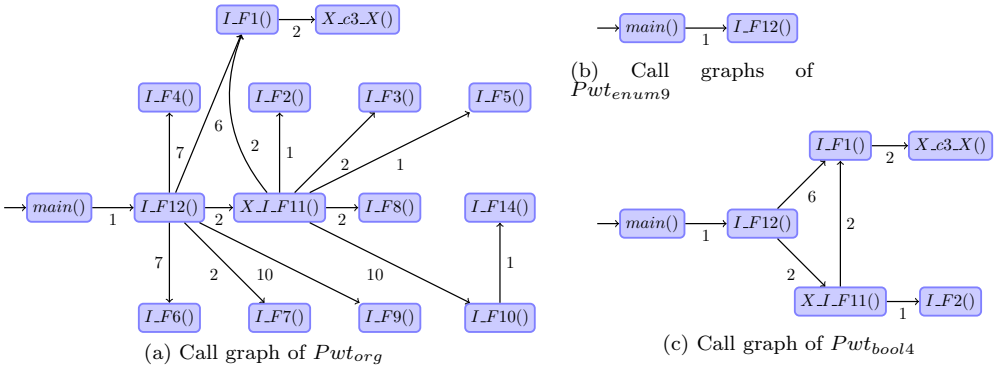
using CBMC solely. The experiments are performed with a desktop computer equipped with CBMC version 5.0, SAT Solver MiniSat 2.2.0, Frama-C version Fluorine-20130601, NPath and Oclint version 0.6 [7] and which comprises the following settings: 3.4Ghz Intel Core i7-3770 CPU, 16.0 Go of memory (RAM), Windows 7 (64 bits), Linux Ubuntu.

4.1 First Case Study

4.1.1 Functionalities and Code of the Component

A powertrain software component oversees critical operations of physical devices in the powertrain such as engine cooling, gear set and suspension adjustments. The C code of the powertrain software component that we received from an industrial partner is designed following the cyclic executive model. Variable and function names have been obfuscated for legal reasons. A single task named I_F12 is executed every time unit for computing new values of 47 output variables depending on the previous values of the same output variables and the current values of 60 input variables. Task I_F12 calls many other interface functions and data processing functions. Our goal is to derive tests for reaching specific values of the output variables G_Enum9 and G_Bool4. Especially, we wanted to determine six input sequences for leading the controller from its initial state to six partial target states specified with the constraints presented in the first column of Table 2b. The initial state of the powertrain sets every numeral variable to 0 and every Boolean variable to $false$. Function call graphs and metrics for the code versions are given in Figure 2a and Table 2a.

[7] http://docs.oclint.org/en/dev



(a) Call graph of Pwt_{org}

(b) Call graphs of Pwt_{enum9}

(c) Call graph of Pwt_{bool4}

Fig. 2. Function call graphs of the powertrain software component and its slices. An arrow $F1 \xrightarrow{n} F2$ means that a call to $F2$ appear n times in $F1$.

4.1.2 Test Generation with CBMC

As a first attempt to generate tests, we applied CBMC solely on the original code we received from our industrial partner, instrumented as described in Section 3.1. Experts of the powertrain component estimated that the delays for reaching each of the five targets defined with variable G_Enum9 do not exceed 60 time units while the delay for reaching $G\_Bool4 = 1$ does not exceed 120 time units. So, we use these delays to set the bound for unrolling the main loop of the scheduler. The other loops in the code are all bounded explicitly and CBMC can unroll them automatically.

The results of applying CBMC with the original code, denoted by Pwt_{org}, are presented in Table 2b. CBMC, in these settings and within less than 2 hours, is not able to derive any input sequence except for target $G\_Enum9 = 0$ satisfied in the initial state of Pwt_{org}. In the other cases, it runs for several hours before crashing.

4.1.3 Test Generation with the Proposed Approach

The approach consists in slicing the code prior to the test generation phase with CBMC. The results also include an evaluation of code slicing and they are presented in Figure 2 and Table 2. Let us denote by Pwt_{bool4} and Pwt_{Enum9} the slices for Pwt_{org} which preserves the observations of G_Bool4 and G_Enum9 at every time step.

In Figure 2, we observe that the function call graph for Pwt_{Enum9} is smaller than the one of Pwt_{bool4} which in turn is smaller than the one of Pwt_{org}. In Table 2a, we observe that metrics for the automatic sliced code on I_F12 are higher than the ones of the original code. This is because Frama-C modifies the structure of the code prior to slicing. We discuss this issue in more details in Section 4.3.1. The manual slice of the code consists in applying manually the slicing operation made by Frama-C directly on the original code; therefore avoiding the intermediate representation. As a result, metrics for code sliced manually are lower than those of the original code.

We generate tests for the two sliced versions of the code and the results are shown in Table 2b. With the proposed approach, five input sequences are generated with the code sliced automatically and similar results are obtained with the code sliced

Function	Version		NCSL	HVM	CCM	NPath
*I_F*12	Original		196	13166	44	1247983200
	Enum9	Auto	377	16835	134	Overflow
		Manual	125	10350	40	966134304
	Bool4	Auto	386	17207	134	Overflow
		Manual	132	10601	40	966134304
X_c3_X	Original		39	1612	11	147
	Bool4	Auto	37	1297	13	228
		Manual	19	847	9	75
*I_F*1	Original		13	313	3	4
	Bool4	Auto	10	259	3	4
		Manual	13	313	3	4
*I_F*2	Original		6	211	2	2
	Bool4	Auto	9	311	2	2
		Manual	6	211	2	2
*X_I_F*11	Original		13	330	4	4
	Bool4	Auto	5	119	2	2
		Manual	4	96	2	2
*I_F*3	Original		32	1680	5	12
*I_F*5	Original		44	2461	12	544
*I_F*8	Original		23	1213	9	43
*I_F*10	Original		25	1143	10	224

(a) Code metrics of the powertrain software component

Target	Code version	Max. exec. dur.	Test length	Comp. delay (min)
$G\_Enum9 = 0$	Original	60	1	203.72
	Auto	60	1	13.61
	Manual	60	1	8.71
$G\_Enum9 = 1$	Original	60	-	-
	Auto	60	59	14.02
	Manual	60	53	9.15
$G\_Enum9 = 2$	Original	60	-	-
	Auto	60	60	14.05
	Manual	60	59	9.07
$G\_Enum9 = 3$	Original	60	-	-
	Auto	60	60	13.90
	Manual	60	60	8.78
$G\_Enum9 = 4$	Original	60	-	-
	Auto	60	60	14.23
	Manual	60	59	8.41
$G\_Bool4 = 1$	Original	120	-	-
	Auto	120	-	-
	Manual	120	-	-

(b) Results of the test generation for the first case study with the proposed framework. "-" means that the execution duration reached a 2 hours timeout.

Table 2
Code metrics and test generation results for powertrain software component

manually. Target $G\_Bool4 = 1$ was not reached with the two sliced versions.

We observe that test computation time for the manually sliced code is lower than the time for automatically sliced code, which is consistent with the different metrics. Although the metrics for each function in Pwt_{org} are almost always lower than the metrics of the same function in the automatically sliced code, the number of acyclic paths in Pwt_{org} is much larger than the number of paths in Pwt_{bool4}; indeed the total number of acyclic paths can be approximated by the product of the number of acyclic paths of functions along the path in the call graphs. Consequently, our results are sound.

4.2 The Second Case Study

4.2.1 Functionalities and Code of the Component

The second case study, as the first one, is designed as a cyclic executive and has also been obfuscated. The code has four tasks *Init1*, *Init2*, *FXXX_HF* and *FXXX_LF*. The first two are called in this order at the beginning of the execution while *FXXX_HF* is called every 50 ms and *FXXX_LF* is called every 1000 ms. The precise scheduling time of these functions is irrelevant to our analysis so time is simplified by using time limits instead. Therefore, *FXXX_HF* is called every time unit while *FXXX_LF* is called every 20 time units. The code has 87 output variables

Target	Code version	Max. execution duration	Test length	Computation. delay (min)
$GlblEnum1\_5 = e1\_5$	Original	50	44	19.12
	Auto slice	50	13	30.77
	Manual slice	50	13	18.63
$GlblEnum1\_5 = e1\_7$	Original	50	50	19.55
	Auto slice	50	50	30.56
	Manual slice	50	50	18.54
$GlblEnum1\_5 = e1\_12$	Original	50	3	19.13
	Auto slice	50	6	30.20
	Manual slice	50	25	17.32
$GlblEnum1\_5 = e1\_14$	Original	50	50	18.76
	Auto slice	50	50	29.85
	Manual slice	50	50	17.33
$GlblBool16 = true$	Original	1320	-	-
	Auto slice	1320	-	-
	Manual slice	1320	-	-
$GlblBool22 = true$	Original	50	21	18.57
	Auto slice	50	41	29.20
	Manual slice	50	41	17.32

Table 3
Results of the test generation for the second case study with the proposed framework. "-" means that the execution duration reached a 2 hours timeout.

and 32 input variables. Once again, our goal is to compute an input sequence for six targets that are presented in Table 3. The initial state of the component sets every numeral variable to 0 and every Boolean variable to *false*. This code has 99 functions and many more calls than the first case study.

4.2.2 Test Generation with CBMC and The Proposed Approach

As for the first case study, we started by applying CBMC alone on the provided code and targets. The results of this experiment are presented in Table 3. In this case study, one loop is not explicitly bounded. A quick manual analysis of the code shows that the loop will be executed at most 5 times. This information was therefore provided to CBMC. After that, using our framework, we applied slicing using Frama-C on the three variables present in the six targets. In this case, the three slicings generate the exact same code except for the variable $GlblEnum1\_5$ and its output function. Since a single variable does not impact metrics and computation time significantly, only one version of the sliced code is used in the following experiments. For this case study, the slicing is not as important and only 5 variables and 10 functions out of the 99 are removed. Moreover, most of the functions removed are small output functions with low impact on the performances. Again, an additional code is produced manually which corresponds to the slicing of Frama-C without intermediate representation. Computation times to generate the input sequence for each targets are presented in Table 3. All targets except $GlblBool16 = true$ were reached in this example. A manual inspection of the code shows that the target cannot be reached within less than 1320 cycles. However, CBMC crashes on

analysing such lengthy executions.

A counter-intuitive result can be observed for this case study. Indeed, it takes more time for CBMC to reach targets on the automatically sliced version of the program than on the original code. However, the manual slice of the program inspired by the results of Frama-C performs better than the original code. Metrics for this example were also calculated but they are not shown due to the high number of functions. Observations are the same as in the first case study : metrics and computation time are closely related and the automatic slicing which is slower than the original code for this example exhibits metrics indicating a higher complexity. We believe this is due to the modifications performed by Frama-C prior to its analysis. CBMC seems to need more cpu time when dealing with code under this form.

4.3 Discussions

4.3.1 Importance of Slicing

Looking at results from the first case study, we see that slicing has a huge impact on performances. Indeed, without slicing, CBMC is not able to find a solution and when slicing is used, it takes only a few minutes to reach five targets. The reduction in the number of paths to explore is therefore important and can make the difference between successful and failed reachability analysis. For the second case study, the slicing is not as important as in the first case study. Therefore, the effect of slicing was not able to outmatch the effect of code modifications from Frama-C. Results from the manual slice, however, show that even in this case, slicing reduces computation time and CBMC performs better on this sliced code than on the original program. It would be possible to replace the slicer in our framework with one that does not modify the code prior to its analysis or modify it differently to always obtain a code that is equally or less complex than its original version.

4.3.2 Determining Minimal Input Sequences

CBMC is a bounded model-checking tool. Therefore a maximum number of instructions must be provided to CBMC for any analysis. This is often done by bounding all loops in the system. In our case, the transition function must be bounded with the number of cycles. This remains an upper bound however and CBMC may find a solution with a lower number of cycles than the maximum provided. Still, the solution does not necessarily represent the minimal input sequence to reach the target. There are no direct means to instruct CBMC to find the minimal solution. However, it is always possible to use a trial and error process by providing different maximum numbers of cycles to find a minimal value and extract the minimal input sequences in terms of number of cycles.

4.3.3 Observations on the Computation Time of CBMC

The computation time of CBMC includes the time for building the SAT formula along with the time for solving the formula. We noticed that the former is higher than the latter and CBMC computes the SAT formula on every analysis. So, given

targets defined with same output variables and reachable within the same execution length, CBMC does not reuse the computed SAT formula which represents the same code used for analysing the reachability of the targets. This does not speed up the process of generating multiple input sequences.

5 Related Work

Test generation for controller software components with test purposes is an active field of research [31,32,28]. The scalability of the test generation approaches with industrial size controllers is an important and difficult issue. Test generation techniques based on random input generation and guided simulation have not been widely accepted by automotive engineers. Techniques based on model-checking, abstract interpretation, static analysis and symbolic execution are quite well-understood and implemented in various tools, e.g., [20,11,28,13,15,5,26,7,36,32]. However, trying to apply those tools to industrial size controllers led to the conclusion that complex code needs to be simplified to facilitate the use of model-checking and symbolic execution techniques for test generation. Abstract interpretation [14,15] and acceleration techniques [6,3] consider abstract semantics of code to ease the code analysis, and therefore to speed up the model-checking process. However, the latter techniques are based on the choice of "good" abstractions of code (coarse abstractions may cause the generation of false-positives and wrong inputs), which is not an easy task even for highly qualified engineers. On the other hand, code slicing which consists in reducing the size and the complexity of the code while preserving certain interesting instructions is advocated as a technique which can be used more easily and without specific knowledge to verify the solution. For this reason, we believe it should preferred to abstract interpretation and acceleration techniques for the test generation with automotive systems.

Most of model-checking or symbolic execution techniques reduce test generation to a SAT/SMT problem. The main difference between model-checking and symbolic execution techniques relies on the method for constructing the formulas. Model-checkers build the formula by translating the whole code without exploring the execution paths while symbolic execution tools like KLEE [7] merge pieces of formulas derived by successive exploration of the execution paths. The complexity of the techniques for building the formulas depends on the features used in code and loop unrolling is usually unavoidable. However, some more recent work [22] addressed the issue of loop unrolling.

So, a test generation approach based on bounded model checking, solvers and slicing was developed and implemented in various frameworks which integrate off-the-shelf tools [36,9,34,8,28]. The scalability of such frameworks in case studies is often characterized in terms of the number of lines of code (LOC) of controllers [9,34,8], however, model-checking engines are more sensible to the number of variables, assignments, decisions and execution paths than to LOC. This is especially the case for engines which reduce the model-checking problem to a satisfiability of propositional logic formulas built from variables in code. Indeed, path explosion is

the impediment to frameworks exploring controller states and execution paths. The higher is the number of assignments along a path, the more complex the formulas encoding the paths and the satisfiability checking will be. SANTE [8] integrates Frama-C [16] and PathCrawler [36] to determine inputs for covering all the execution paths of general C programs. PathCrawler is designed for the coverage of execution paths in programs and is not adapted to generate tests leading to specific target states. ChainCover [32] is a tool for generating and chaining input sequences with implementation of reactive systems with C code. ChainCover which uses CBMC [11] to determine input sequences for covering parts of C code was used for automotive controller code. The report on the case studies in [32] characterizes the complexity of the controller code in terms of the number of different variable types (integer, Boolean, floating point), but not in terms of the number of execution paths. ChainCover does not explicitly slice the code prior to the test generation, expecting that the direct use of CBMC would scale to industrial size controllers. The tool AutoMOTGen [28] determines inputs to cover components of Simulink/StateFlow models of controllers. AutoMOTGen uses sal-bmc model-checker to generate the inputs from sliced versions of SAL programs which are automatically generated from Simulink/StateFlow models. Using sal-bmc to solve the test generation problem requires C programs be translated into SAL programs. Our approach avoids this non-trivial translation.

6 Conclusion

Our main contribution is a framework for generating test cases that are reaching target states in cyclic executive systems. Our framework is composed of a code slicing engine and a bounded model-checking test generation engine. Both engines instrument code prior to the analysis with Frama-C and CBMC. Results from two case studies show that our framework can be efficiently used in an industrial context and can relieve testers from tedious tasks.

Results also show that the use of slicing can help to reduce computation time or even make the difference between a failed search and a successful search in the context of embedded industrial systems. Therefore, slicing makes the code more amenable to model-checking based techniques. Code metrics were used to describe the complexity of the code and better understand the effect of slicing. However, code preprocessing done by the slicing tool used in this analysis caused problems when the amount of removed code was less important. Indeed, code modifications done by the tool would increase test generation time. Nonetheless, we consider slicing to be useful for our specific problem statement. Performing a slicing without code modifications proved better than the original code for all targets in the two case studies. We think that our observations on the importance of slicing are suited for symbolic execution which suffers from the very high number of paths to explore [25].

We believe that the proposed test generation framework can be used in production and it can be augmented with automatic techniques for revealing unbounded loops and over approximation of their bounds. We also plan to study the possibility

to exploit user knowledge including intermediate targets to solve the most difficult targets. Moreover, computing SAT/SMT formulas on each analysis is time consuming and not efficient when targets rely on the same variables and are reachable within same maximal execution duration. Pre-processing and saving parametrized formulas for different targets may help in accelerating the test generation. Future work goes in this direction; we are exploring abstract interpretation and symbolic execution techniques.

References

[1] Arzen, K.-E., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson and L. Sha, *Integrated control and scheduling*, Technical report, Dept. Automatic Control, Lund Institute of Technology (1999), research report ISSN 0820-5316.

[2] Baker, T. P., *The cyclic executive model and ada*, The Journal of Real-Time Systems **1** (1989), pp. 120–129.

[3] Bardin, S., A. Finkel, J. Leroux and L. Petrucci, *Fast: acceleration from theory to practice*, International Journal on Software Tools for Technology Transfer **10** (2008), pp. 401–424.

[4] Barraclough, R. W., D. Binkley, S. Danicic, M. Harman, R. M. Hierons, À. Kiss, M. Laurence and L. Ouarbya, *A trajectory-based strict semantics for program slicing*, Theoretical Computer Science **411** (2010), pp. 1372 – 1386.

[5] Beyer, D., T. A. Henzinger, R. Jhala and R. Majumdar, *The software model checker blast: Applications to software engineering*, International Journal on Software Tools for Technology Transfer **9** (2007), pp. 505–525.

[6] Boigelot, B. and P. Wolper, *Symbolic verification with periodic sets*, in: D. L. Dill, editor, *Proceedings of 6th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science **818** (1994), pp. 55–67.

[7] Cadar, C., D. Dunbar and D. R. Engler, *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs*, in: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[8] Chebaro, O., P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams and B. Yakobowski, *Behind the scenes in sante: a combination of static and dynamic analyses*, Automated Software Engineering **21** (2014), pp. 107–143.

[9] Chimdyalwar, B., *Survey of array out of bound access checkers for c code*, in: *Proceedings of the 5th India Software Engineering Conference* (2012), pp. 45–48.

[10] Clarke, E., A. Biere, R. Raimi and Y. Zhu, *Bounded model checking using satisfiability solving*, Form. Methods Syst. Des. **19** (2001), pp. 7–34.

[11] Clarke, E., D. Kroening and F. Lerda, *A tool for checking ANSI-C programs*, in: K. Jensen and A. Podelski, editors, *Proceedings of the 10th Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science **2988** (2004), pp. 168–176.

[12] Clarke, E. M., Jr., O. Grumberg and D. A. Peled, "Model Checking," MIT Press, Cambridge, MA, USA, 1999.

[13] Cordeiro, L., B. Fischer and J. Marques-silva, *Smt-based bounded model checking for embedded ansi-c software*, in: *Proceedings of the 10th International Conference on Automated Software Engineering*, 2009, pp. 137–148.

[14] Cousot, P. and R. Cousot, *Abstract interpretation: Past, present and future*, in: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science* (2014), pp. 2:1–2:10.

[15] Cousot, P., R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *Combination of abstractions in the astrée static analyzer*, in: *Proceedings of the 11th Asian Computing Science Conference* (2006), pp. 272–300.

[16] Cuoq, P., F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski, *Frama-c: A software analysis perspective*, in: *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (2012), pp. 233–247.

[17] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *An efficient method of computing static single assignment form*, in: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1989), pp. 25–35.

[18] de Moura, L. and N. Bjørner, *Satisfiability modulo theories: introduction and applications*, Communication of the ACM **54** (2011), pp. 69–77.

[19] Ferrante, J., K. J. Ottenstein and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems **9** (1987), pp. 319–349.

[20] Fraser, G., F. Wotawa and P. E. Ammann, *Testing with model checkers: A survey*, Software Testing Verification and Reliability **19** (2009), pp. 215–261.

[21] Horwitz, S., T. Reps and D. Binkley, *Interprocedural slicing using dependence graphs*, ACM Transactions on Programming Languages and Systems **12** (1990), pp. 26–60.

[22] Kersten, R., S. Person, N. Rungta and O. Tkachuk, *Improving coverage of test cases generated by symbolic pathfinder for programs with loops*, ACM SIGSOFT Software Engineering Notes **40** (2015), pp. 1–5.

[23] Kroening, D. and M. Tautschnig, *Cbmc – c bounded model checker*, in: E. Ábrahám and K. Havelund, editors, *Proceedings of the 20th International Conference Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science **8413**, Springer Berlin Heidelberg, 2014 pp. 389–391.

[24] Locke, C., *Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives*, Real-Time Systems **4** (1992), pp. 37–53.

[25] Ma, K.-K., K. Y. Phang, J. S. Foster and M. Hicks, *Directed symbolic execution*, in: *Proceedings of the 18th International Conference on Static Analysis* (2011), pp. 95–111.

[26] MathWorks, T., *Polyspace*.
URL http://www.mathworks.com/products/polyspace

[27] Mathworks, T., *Simulink design verifier 1: User's guide*, Technical report (2012-2013).

[28] Mohalik, S., A. A. Gadkari, A. Yeolekar, K. Shashidhar and S. Ramesh, *Automatic test case generation from simulink/stateflow models using model checking*, Software Testing, Verification and Reliability **24** (2014), pp. 155–180.

[29] Nejmeh, B. A., *Npath: A measure of execution path complexity and its applications*, Communication of the ACM **31** (1988), pp. 188–200.

[30] Ottenstein, K. J. and L. M. Ottenstein, *The program dependence graph in a software development environment*, ACM SIGPLAN Notices **19** (1984), pp. 177–184.

[31] Peleska, J., A. Honisch, F. Lapschies, H. Löding, H. Schmid, P. Smuda, E. Vorobev and C. Zahlten, *A real-world benchmark model for testing concurrent real-time systems in the automotive domain*, in: B. Wolff and F. Zaïdi, editors, *Testing Software and Systems*, Lecture Notes in Computer Science **7019**, Springer Berlin Heidelberg, 2011 pp. 146–161.

[32] Schrammel, P., T. Melham and D. Kroening, *Chaining test cases for reactive system testing*, in: *Proceedings of 25th International Conference on Testing Software and Systems*, Lecture Notes in Computer Science **8254** (2013), pp. 133–148.

[33] Staiger, S., G. Vogel, S. Keul and E. Wiebe, *Interprocedural static single assignment form*, in: *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007.

[34] Venkatesh, R., U. Shrotri, P. Darke and P. Bokil, *Test generation for large automotive models*, in: *Proceedings of the IEEE International Conference on Industrial Technology*, 2012, pp. 662–667.

[35] Weiser, M., *Program slicing*, in: *Proceedings of the 5th International Conference on Software Engineering* (1981), pp. 439–449.

[36] Williams, N., B. Marre, P. Mouy and M. Roger, *Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis*, in: M. Dal Cin, M. Kaâniche and A. Pataricza, editors, *Proceedings of the 5th European Dependable Computing Conference*, Lecture Notes in Computer Science **3463**, Springer Berlin Heidelberg, 2005 pp. 281–292.

[37] Yu, S. and S. Zhou, *A survey on metric of software complexity*, in: *Proceedings of the 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 352–356.