Temporal Property-Based Testing of a Timed C Compiler using Time-Flow Graph Semantics

Saranya Natarajan
EECS and Digital Futures
KTH Royal Institute of Technology, Sweden
saranyan@kth.se

David Broman
EECS and Digital Futures
KTH Royal Institute of Technology, Sweden
dbro@kth.se

***Abstract*—The correctness of a real-time system depends both on it being logically sound and temporally correct. To guarantee temporal correctness, the development of such systems includes: (i) developing a model, (ii) formally verifying the model, and (iii) implementing the verified model using a programming language. The temporal correctness then depends on correctly implementing the model using a real-time programming language and compiling it to a hardware platform. Although the timing semantics of many real-time programming languages are well defined, there is no guarantee that the timing semantics of such programs are correctly translated by the compiler. In this paper, we propose a new method for temporal property-based testing. The general method is implemented and evaluated on the Timed C real-time programming language. We formalize the temporal core semantics of Timed C and then use this formalization to specify the properties that are tested by the new property-based testing tool. More specifically, the tool consist of two parts: (i) a generator that randomly generates Timed C programs, and (ii) a property checker that checks whether the language's timing semantics are correctly captured in its execution. We evaluate the method and tool on an embedded Raspberry Pi platform.**

I. Introduction

The correct implementation of a real-time system depends both on its logical and temporal correctness. In safety-critical systems with temporal constraints, a failure of not meeting one of these constraints can lead to severe damage and may risk human lives. As a consequence, rigorous verification is an essential part when developing safety-critical systems.

The first step in such verification and development process typically consists of the development of a model. There exist many formalisms for modeling dynamic systems, such as timed automata [1] and timed petri nets [2]. In the second step, verification tools such as Uppaal [3] and Kronos [4] are used to formally verify these models. In the third step, the verified models are implemented using a programming language. Today, there exist several programming languages—such as Esterel [5], LUSTRE [6], Ada [7], and RTSJ [8]—with explicit constructs for real-time programming. In the fourth and final step, the source code is compiled to the target machine. However, a compiler is a complex software that implements many algorithms and procedures. A bug in the compiler can result in a bug in the compiled code, something highly unexpected and undesirable. In such scenario, a formally verified system can still fail, even if the original model was verified as being correct.

Proving the correctness of the functional aspects of a compiler is not new: formally verified compilers exist for the C programming language (the CompCert project [9]), functional languages (the CakeML project [10]), and for synchronous languages (a compiler for Lustre [11]).

Although formally verified compilers give high confidence of correctness, the engineering effort of developing a verified compiler is substantial. An alternative is to follow a more traditional test-based approach. Compiler testing tools are widely used when testing compilers and such tools have found many bugs in frequently used compilers [12]. In recent years, property-based testing [13] has gained significant attention and testing tools are available for almost any modern programming language. In contrast to unit-based testing, massive amounts of randomly generated tests are checked if they conform to certain properties. Despite its usefulness, property-based testing has only recently been used in compiler testing [14]. However, to the best of our knowledge, property-based testing has so far not been explored in the context of compilers for languages that support *temporal semantics* and *real-time programming primitives*.

In this paper, we develop a new method for property-based testing of languages and compilers with temporal constraints. The work is focused on a recently proposed real-time programming language called Timed C [15]. Timed C consists of a small set of timing primitives added to the C programming language. Timed C has not been formally specified before and in the first part of this paper, we formalize the temporal aspects of a subset of Timed C using *time-flow graphs* (TFG). We use TFGs to randomly generate Timed C programs that are checked against fundamental properties of the TFG semantics. That is, by using the proposed approach of temporal property-based testing in combination with TFGs, we show how certain timing bugs in an open-source compiler for Timed C can be discovered and resolved. In summary, we make the following contributions:

- We introduce the concept of *time-flow graphs*. We define its high-level operational semantics and use it to describe the semantics of a subset of Timed C (Section III).

- We present the design and implementation of *temporal property-based testing*. This is—to the best of our knowledge—the first work on temporal property-based testing of compilers for real-time systems (Section IV).
- We evaluate our temporal property-based tool by testing the correctness of the KTC compiler, the currently publicly available compiler for Timed C. The evaluation is performed on Raspberry Pi (Section V).

II. Background

In this section, we first discuss the various features of Timed C and then provide a brief introduction to property-based testing.

A. Timed C

Timed C [15] is a recent language for programming real-time systems. Timed C extends the C programming language by introducing temporal and concurrent constructs. The language is used in an on-board student satellite project [15] and there exits an open source end-to-end toolchain supporting measurement-based timing analysis, schedulability analysis, and sensitivity analysis [16].

Timed C is based on the concept of *timing points* (TP). The timing constraints of a Timed C program are specified using timing points. From a programmer's viewpoint, time only elapses at the timing points, i.e., conceptually, executing the code between timing points takes zero time. Currently, Timed C supports soft and firm timing constraints specified using *soft timing points* (STP) and *firm timing points* (FTP), respectively.

```
stp(expr1, expr2, n);
ftp(expr1, expr2, n);
```

Expressions `expr1` and `expr2` represent the relative arrival time and relative deadline of the code fragments succeeding and preceding the TP, respectively. The unit for the delay interval is specified using `n` where the resolution is 10^n seconds. Note that we can also use standard definitions, such as `ms` for milliseconds and `us` for microseconds.

Fig. 1. lists a Timed C program that implements a *generalized multiframe* (GMF) task [17] and its execution diagram. The program consist of two frames where the first frame (lines 2-3) has a relative arrival time of 15 ms and a soft

```
1 task foo(){
2   while(1){
3     bar();
4     stp(15,10,ms);
5     far();
6     ftp(15,10,ms);
7   }
8 }
```

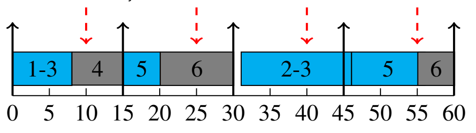


Fig. 1: A Timed C program implementing a generalized multiframe task with soft and firm timing points. The black upward arrows depict arrival times and the red downward arrows depict the deadlines.

```
1 void compute(int* a){
2   int b[100];
3   init(a);
4   while(1){
5     improve(b,a);
6     critical{
7       memcpy(a,b,sizeof(int)*100);
8     }
9   }
10  ftp(200,200,ms);
11 }
```

Fig. 2: A Timed C program implementing an anytime algorithm using the `critical` construct.

deadline of 10 ms, as specified by the `stp` on line 4. Similarly, the `ftp` on line 6 enforces the second frame (line 5) to have a firm deadline of 10 ms and has a relative arrival time of 15 ms.

In the timing diagram, task `foo` starts executing at $t = 0$ and function `bar()` completes at $t = 8$. The `stp` at line 4 then delays the execution until $t = 15$. Function `far()` starts executing at $t = 15$ and completes at $t = 20$. The `ftp` delays the next call to `bar()` until $t = 30$. In the second iteration of the `while` loop, `bar()` arrives at $t = 30$. Its absolute deadline is at $t = 40$. Note that although `bar()` arrives at $t = 30$ it starts only at $t = 32$ due to some platform dependent *release jitter*. Since this instance has a soft deadline, the `stp` permits overrun and `bar()` completes at $t = 46$. Note that although the previous frame overran its execution until $t = 46$, the next frame still arrives at $t = 45$, which is 15 ms since the arrival of the previous frame, as depicted by the black arrow. The next instance of `far()` starts executing at $t = 46$ and has not completed at $t = 55$. The `ftp` construct enforces a firm deadline by aborting `far()` at its absolute deadline $t = 55$.

In some scenarios, aborting a computation may result in unexpected behaviour. Timed C's `critical` construct is used to prevent such interruption. We illustrate an example program using `critical` in Fig. 2. This code is based on an example from [15] that implements an *anytime algorithm*. Anytime algorithms [18] support approximate computation where the quality of the solution improves with increasing execution time. In Fig. 2, function `improve()` implements an anytime algorithm and the `ftp` (line 10) makes it possible to interrupt it. Note that the function call to `init()` (line 3) computes and stores an unoptimized initial solution in a. The function `improve()` improves a by computing and storing a new solution in b. The code on line 7 then copies b to a. To ensure that this copying is not interrupted by the `ftp`, the `memcopy` is enclosed within the `critical` construct.

B. Property-based testing

Property-based testing (PBT), also known as Quickcheck [13], replaces handwritten tests with automatic and random test generation. It consists of two components: (i) the *property* component and (ii) the *generator*. The *property* component captures the specification of the program and the *generator* randomly generates arbitrary input. A PBT tool generates large number of random inputs, and tests these inputs on a set of specified properties. Consider a simple example of a function `square()` that takes as input an integer and computes its square. For input `x` the function computes square using `y=x*x` and returns `y`. To test if this function is correct, we specify a property as `square(x) = x*x`. The generator generates random values of type `int` and the function under test executes every input. Each test case is tested for the specified property. Note that each instance is a test case and a failure results in the generation of a counterexample.

Suppose `square()` is compiled using two compilers, A and B. Moreover, assume that A correctly implements operator precedence while B does not. This results in square operation evaluated as `y=(x*x)` and `(y=x)*x` in A and B, respectively[1]. When applying property-based testing, there are no failures using A, whereas B results in that all tests fail.

In this paper we present a temporal property-based testing methodology and use it to test the correctness of KTC—a Timed C source-to-source compiler. The KTC compiler compiles Timed C programs to platform specific C code. The generated C code is then compiled using an off-the-shelf compiler. The novelty of our method lies in the fact that we present a PBT method to test KTC with respect to how it translates the *temporal properties* of the input program.

III. Time Flow Graph

In this section, we introduce the concept of *time-flow graphs* (TFG), as a way to define the high-level semantics of Timed C. Note that we need a formal definition of the semantics to be able to test the correctness of the compiler. Note also that TFG formalizes *only* the temporal properties of Timed C and abstracts away all other details.

A. The TFG Definition

A TFG is a formal representation of a Timed C task[2]. The TFG is a directed graph where the vertices of the graph encode the timing points and code fragments. For instance, Fig. 3(b) is the TFG of the Timed C task in Fig. 3(a). A definition of TFG is listed in Fig. 4. It consists of five components: (i) *graph structure*, (ii) *program properties*, (iii) *platform dependent parameters*, (iv) *states*, and (v) *evaluation functions*. In the rest of this section, we describe each of these five components.

Graph Structure: In Fig. 3, the TPs and code fragments are formally defined as $P = \{p_0, p_1, p_2\}$ and $F = \{f_1, f_2\}$. The TPs p_0, p_1, and p_2 represent the start of the task at line 1, the STP at line 4, and the FTP at line 8, respectively. Similarly, the code fragments f_1 and f_2 correspond to the code on lines 2-3 and lines 5-7, respectively. As illustrated in Fig. 3, a circle is a TP and a square is a code fragment. The *initial timing point* (p_0) that encodes the start of a task has no incoming edge. This is represented using a filled circle with arrow.

Program Properties: The various timing constraints of real-time programs are specified using timing parameters. The timing parameters can be broadly divided into (i) *platform dependent* and (ii) *platform independent*. In the definition of the TFG, we group together platform independent timing parameters that can be directly derived from the Timed C program under *program properties*. For instance, in Fig. 3, the program properties of the TPs in the TFG are, $t_A(p_0) = 0$, $t_D(p_0) = 0$, $t_A(p_1) = 30\,\text{ms}$, $t_D(p_1) = 20\,\text{ms}$, $k(p_1) = \texttt{soft}$, $t_A(p_2) = 20\,\text{ms}$, $t_D(p_1) = 10\,\text{ms}$, and $k(p_2) = \texttt{firm}$.

Platform Dependent Parameters: The functions t_j, t_r, t_B, and t_W are used to annotate the TFG with platform dependent parameters. These timing parameters cannot be directly derived from the Timed C code. The release jitter of a TP can be computed using platform specific tests such as `cyclictest` tool for a Linux system. Trigger precision is defined as the extra time taken to jump out of the critical block. A measurement-based approach to compute trigger precision, BCET, and WCET of a code fragment is proposed in [16].

States: As the program executes, it constantly updates its *program state* and *machine state*. The *program state* contains information pertaining to the data stored in the various variables of the program. The machine state, M, of a program execution contains information about the absolute time elapsed since the start of the program, the machine's cache state, etc.

Evaluation Functions: The functions $\mathit{critical}$, vnext, fnext, and $\mathit{execute}$ are the set of evaluation functions in the TFG. These function are used to define the semantics of the TFG.

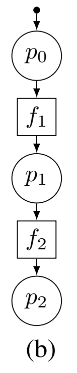
The function $\mathit{critical}$ maps a code fragment enclosed within a `critical` construct to `true`. For example in Fig. 3, $\mathit{critical}(f_2)$ is `true` while $\mathit{critical}(f_1)$ is `false`. The function fnext takes a code fragment, f, as input and returns a FTP, p, if there exists a path from f to p such that there exist no other TP in the path from f to p. Otherwise, it returns $\perp$.

The function $\mathit{execute}$ captures the actual execution of a code fragment. It takes as input a code fragment f, program state z, machine state m, and a real time d. Here d is the real time at which a timer is set to expire. If $\mathit{critical}(f) = \texttt{false}$ a timer expiry will abort the execution of f. The function returns the updated program state z', machine state m', and the status of the timer expiry where `true` indicates timer expiry.

The function delay maps the current program state, machine state, and next absolute arrival time to a new machine state. The delay function is responsible for delaying the execution

```
1 task far(int* a){
2   int b[100];
3   boo(b);
4   stp(30,20,ms);
5   critical{
6     foo(a,b);
7   }
8   ftp(20,10,ms);
9 }
```

(a)



(b)

Fig. 3: A Timed C program with both `stp` and `ftp`.

[1]Surprisingly, the latter expression compiles using a modern compiler, such as GCC, and gives a warning message stating that the result of `*x` is unused.

[2]The semantics of Timed C has changed sligtly since the original paper [15] was published. The semantics described in this paper is based on the compiler used in the end-to-end toolchain by Natarajan *et al.* [16].

Graph Structure:

- P is the set of timing points.
- F is the set of code fragments.
- $V = P \cup F$ is the set of vertices where $P \cap F = \varnothing$.
- $E = V \times V$ is the set of edges.

Program Properties:

- $k : P \to \{\texttt{soft}, \texttt{firm}\}$ is a function that maps a TP to its *kind* of real-time deadline.
- $t_A : P \to \mathbb{T}$ maps a TP to its relative *arrival time*.
- $t_D : P \to \mathbb{T}$ maps a TP to its relative *deadline*.

Platform Dependent Parameters:

- $t_j : P \to \mathbb{T}$ maps a TP to its maximal *release jitter*.
- $t_r : P \to \mathbb{T}$ maps a TP to its *trigger precision*.
- $t_B : F \to \mathbb{T}$ maps a code fragment to its best-case execution time (BCET).
- $t_W : F \to \mathbb{T}$ maps a code fragment to its worst-case execution time (WCET).

States:

- Z is the set of program states.
- M is the set of machine states.

Evaluation Functions

- $\textit{critical} : F \to \mathbb{B}$ maps a code fragment enclosed within critical block to `true` and a non-critical block to `false`.
- $\textit{vnext} : Z \to V$ maps a program state to the next TP or the code fragment the program jumps to during its execution.
- $\textit{fnext} : F \to \{\bot\} \cup P$ maps a code fragment to a TP if there exists a path with no other TP from F to a FTP and $\bot$ if no such path exists.
- $\textit{execute} : F \times Z \times M \times \mathbb{T} \to Z \times M \times \mathbb{B}$ takes as input a code fragment, a program state, a machine state, and the real time. It returns the new program state, a machine state, and a boolean value that results from executing F. The boolean value is `true` if the timer expired during this execution,
- $\textit{delay} : P \times M \times \mathbb{T} \to M$ maps a TP, a machine state, and real time to a machine state. It is responsible for delaying the execution of a program.
- $t_c : M \to \mathbb{T}$ maps machine state to the real time which is the absolute time elapsed since start.

Fig. 4: Definition of time-flow graphs. Note that we only formalize a high-level abstraction of the semantics, where we abstract away the functional aspects, such as execution of code in program fragments.

of the program. The *delay* function returns immediately if the time elapsed since the start of the program (real time) is greater than or equal to the specified next absolute arrival. Otherwise, it delays the execution of the program until the real time and specified absolute arrival time become equal. The function t_c maps the machine state to the real time. Here, the real time is total time elapsed since the start of the program.

B. Operational Semantics

In this subsection, we present the semantics of the TFG using *inference rules*, as is standard when describing small-step operational semantics [19]. In Fig. 5 we formally define the timing semantics of TFGs using the *transition relation* $(v, z, m, a) \longrightarrow (v, z, m, a)$, where $v \in P \cup V$ is either a timing point $p \in P$ or a code fragment $f \in F$, $z \in Z$ the program state, $m \in M$ the machine state, and $a \in \mathbb{T}$ the absolute arrival time. We define one transition rule for TP (`T-TP`) and three different transitions for code fragments (`T-SOFT`, `T-FIRM1`, and `T-FIRM2`). Note that at any step only one rule applies. We explain the transition rules using the example in Fig. 6. In our discussion, we use t to represent the real time elapsed since the start. We consider the machine state at $t = n$ to be m_n.

Function `action()` in Fig. 6 implements an example Timed C program with both STP and FTP. We start the transition at p_0 from state $(p_0, z, m, 0)$. The vertex p_0 is a TP and the only rule that applies is `T-TP` . Rule `T-TP` has 3 premises. The first premise identifies the outgoing edge from the TP which is used to determine the next vertex in the transition. The second premise computes the absolute arrival time of the TP which is used by the *delay* function. Finally the third premise calls *delay* which delays the execution until the real time becomes equal to the computed absolute arrival. In Fig. 6, the only outgoing edge from p_0 is (p_0, f_1). Function *delay* returns m_0 at $t = 0$. Hence, applying `T-TP` updates the TFG state to $(f_1, z, m_0, 0)$. Note that the program state remains unaltered because a TP does not change the value of any program variable, it only affects the real time encoded in the machine state.

The next transition from $(f_1, z, m_0, 0)$ will either apply `T-SOFT`, `T-FIRM1`, or `T-FIRM2` because $f_1 \notin P$. In all these rule we start by computing $\textit{fnext}(f_1)$. Since there exists a path from f_1 to p_1 and $k(p_1) = \texttt{soft}$ we get $\textit{fnext}(f_1) = \bot$. Hence, the only valid transition rule is `T-SOFT`. The second premise of `T-SOFT` is responsible for the execution of f_1 which corresponds to executing line 2 of the Timed C program. Note that in `T-SOFT` the *execute* function is called with ∞ which means the timer is set to expire at ∞. We specify a timeout of ∞ to encode the non firm timing constraint of the code fragment. From the execution diagram in Fig. 6 we see that line 2 completes at $t = 8$ which implies that the timer did not expire. Hence, *execute* returns $(z_1, m_8, \texttt{false})$. In this example, z is updated because of variables in `init`. The third premise of `T-SOFT` computes p_1 as the next vertex and updates the TFG state to $(p_1, z_1, m_8, 0)$. Let us further look at transition from $(p_1, z_1, m_8, 0)$. Since $p_1 \in P$, we apply `T-TP`. The first premise computes the next vertex as f_2 and the second premise computes $a' = 10$ ($a = 0$ and $t_A(p_1) = 10$). As depicted in Fig. 6 the *delay* function starting at $t = 8$ delays the execution of the program until $t = 10$. This updates the TFG state to $(f_2, z_1, m_{10}, 10)$.

$$\boxed{(v,z,m,a) \longrightarrow (v,z,m,a)}$$

$$\frac{(p,v)\in E \quad a' = a + t_A(p) \quad m' = \mathit{delay}(p,m,a')}{(p,z,m,a) \longrightarrow (v,z,m',a')}\ \text{[T-TP]}$$

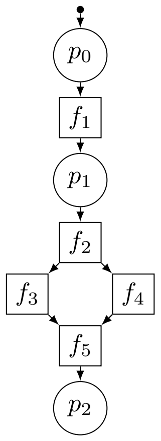
$$\frac{\bot = \mathit{fnext}(f) \quad (z',m',\texttt{false}) = \mathit{execute}(f,z,m,\infty) \quad v = \mathit{vnext}(z')}{(f,z,m,a) \longrightarrow (v,z',m',a)}\ \text{[T-SOFT]}$$

$$\frac{p = \mathit{fnext}(f) \quad d = t_D(p) + a \quad (z',m',\texttt{false}) = \mathit{execute}(f,z,m,d) \quad v = \mathit{vnext}(z')}{(f,z,m,a) \longrightarrow (v,z',m',a)}\ \text{[T-FIRM1]}$$

$$\frac{p = \mathit{fnext}(f) \quad d = t_D(p) + a \quad (z',m',\texttt{true}) = \mathit{execute}(f,z,m,d)}{(f,z,m,a) \longrightarrow (p,z',m',a)}\ \text{[T-FIRM2]}$$

Fig. 5: Transition rules defining the operation semantics of TFGs. See Fig. 4 for the definition of TFGs.

```
 1 task action(int c){
 2   init();
 3   stp(10,10,ms);
 4   if(c){
 5     act_one();
 6   }
 7   else{
 8     act_two();
 9   }
10   critical{
11     c=next();
12   }
13   ftp(30,30,ms);
14 }
```



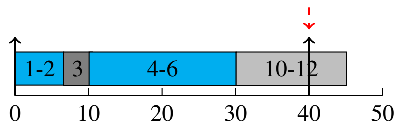


Fig. 6: An example Timed C program, its TFG, and its execution diagram.

For the next transition we have $f_2 \in F$, $\mathit{fnext}(f_2) = p_2$, and $k(p_2) = \texttt{firm}$, Hence, the two potential rules for transition from $(f_2, z_1, m_{10}, 10)$ are `T-FIRM1` and `T-FIRM2`. As discussed previously, the execution of a code fragment with firm deadline is aborted when it misses its deadline, otherwise it executes to completion. We capture these two scenarios using the two transition rules `T-FIRM1` (no deadline miss) and `T-FIRM2` (with deadline miss). Additionally, we use a timer to track the code fragment's deadline in *execute*. Hence, in the second premise (of both `T-FIRM1` and `T-FIRM2`) we compute the absolute deadline of the code fragment which in turn defines the real time at which the timer in *execute* expires.

IV. Temporal Property-Based Testing

In this section we present the methodology and tool for temporal property-based testing (TPBT) using TFGs.

A. Design Overview

In this paper, the main objective of the TPBT is to test Timed C's source-to-source compiler called KTC. More specifically, we test the correctness of KTC's source-to-source compilation w.r.t. Timed C's semantic. As discussed previously, property-based testing consist of a *generator* and a *property component*. In our TPBT, the generator generates TFGs and the property component basically specifies the transition system listed in Fig. 5.

The design overview of our TPBT tool is depicted in Fig. 7. The main idea is to randomly generate many Timed C programs, compile, execute, and verify its correctness. The challenges concern how to (i) generate relevant Timed C programs, and (ii) how to check temporal properties of programs with real-time constraints. Since TFGs capture the essential parts of a Timed C program, we use TFGs both for generating Timed C programs and property specifications. The TFGs are generated in the TFG generation phase. These randomly generated TFGs are then used to generate *instrumented Timed C code* files and *property specification code* in the Timed C generation and TFG property code generation phases, respectively (see Fig. 7). The property specification code is compiled and executed, which produces the *property log* that traces the transition of the TFG. We compile the instrumented Timed C code using KTC to C code. When this C code is compiled and executed using an off-the-shelves C compiler, it produces an *execution log* that traces the program's flow and its temporal aspects. The property checker then takes as input the property log and the execution log and tests if the defined semantics are captured in the program execution. In the following subsections, we discuss these different phases.

B. Random TFG Generation

We have designed and implemented a recursive algorithm for random TFG generation. The algorithm makes use of probability distribution tables to enable the configuration of the random generation. For instance, the user of the tool can specify the percentage of generated STPs, FTPs, fragments, if-then-else conditions, loops, etc. The actual algorithm is left out because of space limitations.

C. Timed C code generation

In this subsection we explain Timed C code generation using the example in Fig. 8. The Timed C code generation assigns unique IDs to the TPs and the code fragments. In Fig. 8, these

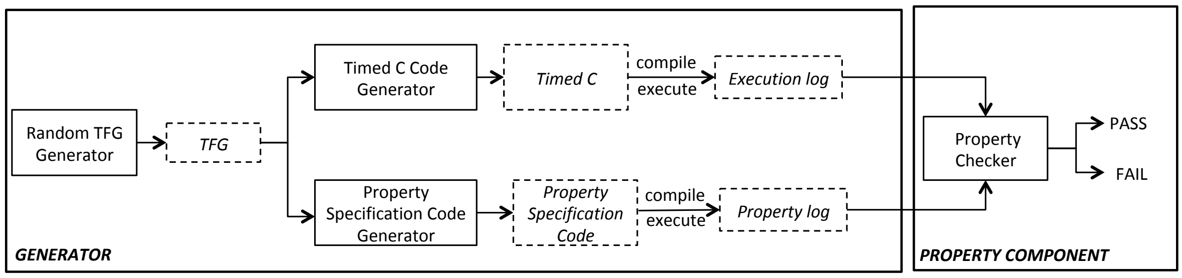
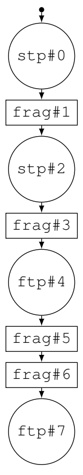


Fig. 7: Overview of the proposed property-based testing tool for Timed C. The tool consists of two main components: a generator and a property checker.



```
1 task fun(){          //stp#0
2   cf1();             //frag#1
3   stp(30,30,ms);//stp#2
4   cf2();             //frag#3
5   ftp(20,20,ms);//ftp#4
6   critical{
7      cf3();          //frag#5
8   }
9   cf4();             //frag#6
10  ftp(30,10,ms);//ftp#7
11 }
```

(a) (b)

Fig. 8: An example TFG and Timed C code with IDs that are generated by the instrumentation.

IDs correspond to the IDs of nodes in the TFG. For instance, line 2 in the Timed C code corresponds to *frag#1* in the TFG. The Timed C code generation inserts six different type of instrumentation instruction: (i) `PBT-ID`, (ii) `PBT-START`, (iii) `PBT-END`, (iv) `PBT-OV`, (v) `PBT-CRITICAL`, and (vi) `PBT-TRIG`. These instructions trace the execution and timing information of the TPs and code fragments in the execution log using the following format:

- `stp#id str fin`
- `ftp#id str fin ov cr tr`
- `frag#id`

where `id` is the unique ID, `str` is the start time of a TP, `fin` is the finish time of a TP, `cr` signals the execution of a critical section, and `tr` is the trigger precision of a critical code fragment.

Fig. 9(a) lists an example execution log of Fig. 8. Matching the lines from the execution logs with the format specified above, we see that `stp#2` starts at $t = 20$ and finishes at $t = 30$. Similarly, `ftp#4` starts at $t = 41$ and finishes at $t = 50$. However, we see `ftp#7` has a start time of $t = 0$. Note that a start time equal to zero indicates an abortion due to a deadline miss (an overrun). This is further confirmed by the fact that `ftp#7` has an overshot due to execution of a critical fragment `frag#5`. Note that due to this overrun `frag#6` is not executed.

D. Property specification code generation

The property specification code generation takes as input a TFG and generates a property specification code. The property specification code when compiled and executed results in the property log. Note that the property log traces the transition of the TFG using the transition rules depicted in Fig. 5 in the following format:

- `stp#id` s_e–s_l f_e–f_l
- `ftp#id` s_e–s_l f_e–f_l
- `frag#id`

where `id` is the unique ID of a TP or code fragment, s_e and s_l are the earliest possible and latest permissible start time of a TP, respectively. Similarly, f_e and f_l are the earliest possible and latest permissible finish time of a TP, respectively.

Fig. 9(b) lists an example property log of Fig. 8. According to the Timed C semantics, the initial TP `stp#0` can start and finish between 0 and ∞. Hence, in line 1 `stp#0` has s_e=f_e=0 and s_l=f_l=∞. Also note that the absolute arrival time of `frag#1` is 0.

Let us consider `ftp#4` with relative arrival of 20 ms. Note that `ftp#4` enforces a firm deadline. Hence, if `frag#3` tries to run endlessly, `ftp#4` will interrupt its execution at its absolute deadline $t = 50$ ms. Hence, s_l of `ftp#4` is 50 ms, which will be checked using the property log. Because `ftp#4` does not allow overruns, its f_l is equal the absolute arrival time of `frag#5`, which is also 50 ms.

In summary, the s_e of a TP is equal to the absolute arrival time of its previous code fragment and its f_e is the absolute arrival time of next code fragment. Since, a STP allows overruns, its s_l and f_l are ∞. An FTP does not allow overruns. Hence, its s_l is equal to the absolute deadline of the previous TP. When the relative deadline is less than the relative arrival time of an FTP, its f_l is equal to the absolute arrival time of next code fragment. Otherwise, its f_l is equal to the absolute deadline of the previous code fragment.

E. Property Checker

As discussed above, the execution log traces the program flow along with the start time, the end time, the overshot, and trigger precision of TPs. By contrast, the property log specifies the program flow along with the allowed intervals of the start and end times. As a consequence, the *property checker* performs

(a)	(b)

`1 stp#0 0 0`
`2 frag#1`
`3 stp#2 20 30`
`4 frag#3`
`5 ftp#4 41 50 0 0 0`
`6 frag#5`
`7 ftp#7 0 80 5 1 5`
	`1 stp#0 0-inf 0-inf`
	`2 frag#1`
	`3 stp#2 0-inf 30-inf`
	`4 frag#3`
	`5 ftp#4 30-50 50-50`
	`6 frag#5`
	`7 frag#6`
	`8 ftp#7 50-60 80-80`

Fig. 9: Examples of (a) an execution log (b) a property log.

testing by comparing the two logs. For example, when `str` is not 0 the property checker checks if the `str` (or `fin`) in the execution log is between s_e(or f_e) and s_l(or f_l) in the property log. Note that the checking algorithm also includes cases not shown in the example. For instance, handling of `stp` overruns that affect the start time of another `ftp`. A detailed explanation has been left out because of space limitations.

V. Evaluation

This section evaluates the correctness of the KTC compiler, followed by a correctness evaluation of the TPBT tool itself.

A. Experimental Setup

The TPBT tool generates the TFGs, the Timed C programs, and property specification codes. We used a random C program generator tool called CSmith [12] 2.3.0 to generate code for the code fragments in the Timed C program. We use KTC to compile the Timed C code to target specific C code and GCC to compile the generated C code. We generate the execution log on Raspberry Pi 2 Model B with ARM Cortex A7 CPU clocked at 900MHz running a Raspbian Linux distribution with the `PREEMPT_RT` kernel and Intel i5 CPU running Ubuntu OS. However, due to space constraint we only show the results from our experiments on Raspberry Pi. The TPBT tool runs on an Intel i5 CPU running Ubuntu OS. A key motivation for using Raspberry Pi in our evaluation is to test the correctness of KTC on a small microcontroller platform.

B. Correctness of the KTC compiler

We tested if the KTC compiler correctly translates the semantics of timing point delay, STP with overrun, and FTP with overrun. We generated 1000 arbitrary valid TFGs with STP, FTP, code fragment, and critical code fragment with probability distribution of 30%, 30%, 30%, and 10%, respectively. A valid TFG satisfies all the TFG properties and passes the KTC static analysis. Note that the C code generated by CSmith does not always run to completion. Hence, we ran a script to disallow generation of CSmith functions that ran endlessly.

The results of our experiment is depicted in Fig. 10, where the blue and red bars indicate the number of TP executed that pass and fail the TPBT, respectively. In Fig. 11, we observe that KTC correctly translates all STPs. However, there are failures in FTP. On analyzing the test inputs we found this occurs in certain scenario due to, (i) FTP not returning the correct overshot and (ii) FTP failing to interrupt execution of code fragment with firm deadline. We fixed this in KTC and

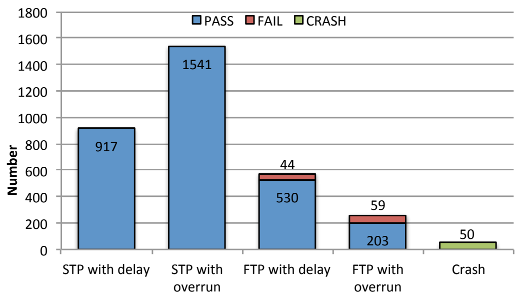


Fig. 10: TPBT testing the KTC compiler.

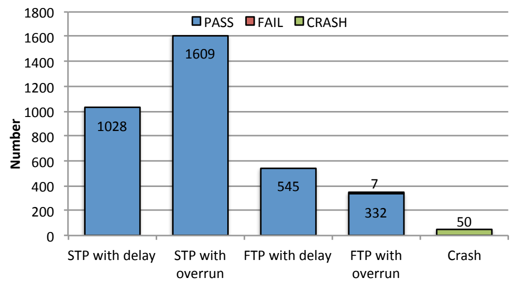


Fig. 11: TPBT testing the fixed KTC compiler.

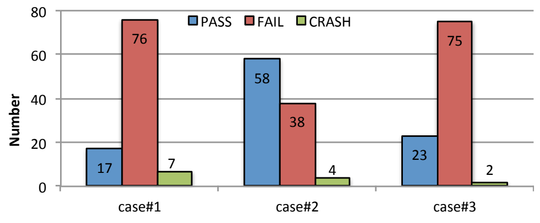


Fig. 12: Testing the correctness of the TPBT tool.

ran the TBPT tool on the same set of inputs. The results with the fixes are depicted in Fig. 11. We currently do not have fixes for 7 of the failing inputs in Fig. 11. The *crashes* in Fig. 10 and Fig. 11 are test inputs that crashed during runtime. Note that the execution log is not generated when the program crashes. Currently we do not know if these crashes are due to KTC translation of Timed C code or generated CSmith functions.

C. Correctness of the TPBT tool

The TPBT tool like every other software implementation is prone to bugs. Hence, we evaluate the correctness of the TPBT tool itself. We achieve this by inserting errors in the KTC compiler and performing property-based testing on these buggy implementations. We tested three cases:

Case#1: If the TPs delay the execution of the program for a time interval less than their relative arrival time, then all inputs that pass the KTC static analyzer and have more than one TP (where at least one TP does not overshot) will fail the TPBT.
Case#2: If FTP does not impose a firm deadline then all inputs that pass the KTC static analyzer and miss a firm deadline will fail the TPBT tool.
Case#3: If the critical construct allows the FTP to interrupt the execution of the critical section, then all inputs that pass KTC static analysis and have a critical code fragment—that otherwise runs endlessly—will fail the TPBT tool.

We tested the above cases by generating 100 inputs for each of them. The results are presented in Fig. 12. Case#1 is valid only for inputs with more than one TP because we do not decrease the arrival time of the initial TP which means that the absolute arrival time of first code fragment is $t = 0$. For case#1, we modified the KTC compiler to decrease the arrival time of the TPs by 5 ms. In Fig. 12, we see that 17 cases passed while 76 failed. On analyzing the results we found all inputs that passed either had no TPs or all TPs were STP with overrun. For case#2, we modified the KTC compiler to not interrupt the execution of a code fragment with firm deadline. In Fig. 12, we see that 58 passed and 38 failed. On analyzing the test results, we found that all inputs that passed had no FTP with overrun. For case#3, we modified the KTC compiler to interrupt the execution of a critical code fragment. In Fig. 12, we see that 23 passed and 75 failed. On analyzing the test result we found that all inputs that passed had no critical code fragment. In summary, in all these test cases, we found that the TPBT tool detected the relevant errors which were introduced.

VI. Related Work

Compiler Testing: A large number of tools and methods for testing compilers has been studied by researchers [12], [20], [21]. CSmith [12], a tool for generating C programs, has been used in randomized differential testing of C compilers. A different approach, now called property-based testing, originally introduced by Claessen and Hughes [13], are now used in many tools for testing various applications [22], [23]. Midtgaard *et al.* [14] propose PBT for testing the back-ends of two OCaml compilers. These different testing tools and techniques for *untimed* programming languages have been successfully used to find bugs in many production compilers. In contrast to previous work, we introduce a temporal property-based testing method to test compilers for *timed* programming languages. The main objective of our method is to test whether the compiler correctly translates the timing semantics of the programming language. To the best of our knowledge, this the first work on *temporal* property-based testing of compilers.

Task Model: Timed automata [1] and the digraph task model [24] are well-established and widely used for modeling real-time applications. However, compared to TFGs, these existing models and formalisms are not close to the semantics of Timed C, making them hard to use in our setting.

VII. Conclusion

In this paper, we first present the concept of time-flow graphs and use it to formalize a subset of Timed C. We use this concept to design and implement a temporal property-based testing tool, to test the correctness of Timed C's source-to-source compiler.

Acknowledgement

We would like to thank Björn B. Brandenburg, Mitra Nasri, and Geoffrey Nelissen for early discussions on the concept of TFGs. We would also like to thank Viktor Palmkvist for ideas on the log generation. This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032).

References

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[2] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed petri nets." Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[3] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International journal on software tools for technology transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[4] S. Yovine, "Kronos: A verification tool for real-time systems," *STTT*, vol. 1, no. 1-2, pp. 123–133, 1997.

[5] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[7] A. Burns and A. Wellings, *Concurrent and real-time programming in Ada*. Cambridge University Press, 2007.

[8] G. Bollella and J. Gosling, "The real-time specification for java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

[9] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reasoning*, vol. 43, no. 4, pp. 363–446, 2009. [Online]. Available: https://doi.org/10.1007/s10817-009-9155-4

[10] Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish, "A new verified compiler backend for CakeML," in *International Conference on Functional Programming (ICFP)*, J. Garrigue, G. Keller, and E. Sumii, Eds. ACM, 2016.

[11] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 586–601.

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[13] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *ACM SIGPLAN Notices*, vol. 46, no. 4, pp. 53–64, 2011.

[14] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson, "Effect-driven QuickChecking of compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–23, 2017.

[15] S. Natarajan and D. Broman, "Timed C: An Extension to the C Programming Language for Real-Time Systems," in *RTAS*, 2018, pp. 227–239.

[16] S. Natarajan, M. Nasri, D. Broman, B. B. Brandenburg, Nelissen, and Geoffrey, "End-to-end temporal sensitivity analysis using timed c," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2010.

[17] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[18] S. J. Russell and S. Zilberstein, "Composing real-time systems." in *IJCAI*, vol. 91, 1991, pp. 212–217.

[19] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[20] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[21] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[22] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing autosar software with quickcheck," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–4.

[23] J. Hughes, B. C. Pierce, T. Arts, and U. Norell, "Mysteries of dropbox: property-based testing of a distributed synchronization service," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 135–145.

[24] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 71–80.