Timed C: An Extension to the C Programming Language for Real-Time Systems

Saranya Natarajan
KTH Royal Institute of Technology
saranyan@kth.se

David Broman
KTH Royal Institute of Technology
dbro@kth.se

Abstract—**The design and implementation of real-time systems require that both the logical and the temporal behavior are correct. There exist several specialized languages and tools that use the notion of logical time, as well as industrial strength languages such as Ada and RTJS that incorporate direct handling of real time. Although these languages and tools have shown to be good alternatives for safety-critical systems, most commodity real-time and embedded systems are today implemented in the standard C programming language. Such systems are typically targeting proprietary bare-metal platforms, standard POSIX compliant platforms, or open-source operating systems. It is, however, error prone to develop large, reliable, and portable systems based on these APIs. In this paper, we present an extension to the C programming language, called Timed C, with a minimal set of language primitives, and show how a retargetable source-to-source compiler can be used to compile and execute simple, expressive, and portable programs. To evaluate our approach, we conduct a case study of a CubeSat satellite. We implement the core timing aspects in Timed C, and show portability by compiling on-board software to both flight hardware, and to low-cost experimental platforms.**

I. Introduction

In real-time systems—such as autonomous aircraft, cars, and robots—the correctness of the system depends on both its logical and temporal correctness. To correctly implement these systems, timing and timeliness of computations need to be both unambiguously *specified* in the software system and correctly *executed* on the real-time execution platform. However, many of the modern programming languages that are used to implement these time-sensitive applications lack explicit semantic constructs to express timing properties, thus making it extremely hard to verify temporal properties or to react systematically to timing violations.

Unsurprisingly, due to the fundamental aspect of this problem, there exists a large body of research work within the real-time and embedded systems community. In synchronous languages, such as Lustre [1], Esterel [2], and Signal [3], time is abstracted into logical ticks. PTIDES [4] is an event-triggered programming model, and Giotto [5] is based on a time-triggered model that separates platform dependent and independent concerns. The key aspect of all these approaches is the separation of *logical time* from *real time*: the programming model is deterministic and a realization of the real-time system conforms to the programming model if the operating system and the execution platform can guarantee that there is a feasible schedule that does not violate any timing constraints. Thus, for the realization of the program to be correct, sound bounds of the worst-case execution time (WCET) [6] must be established prior to scheduling analysis.

In contrast to hard real-time problems that require WCET analysis, several programming models have been proposed on top of existing general purpose languages that can react on timing violations. Ada [7] has explicit support for expressing absolute delays and timeouts. The real-time extension to Java, RTJS [8], supports real-time threads and event handling. Several timing extensions to the C programming language have been proposed in the research literature in the 1990s [9]–[11]. In these programming models, the programmer explicitly reasons about real time.

Although certain languages and systems, such as SCADE [12] and Ada, have been successfully used for safety-critical systems in industry, most commodity real-time systems are today developed using the standard C programming language. Such systems typically target either proprietary bare metal platforms, or real-time operating systems such as FreeRTOS [13] and UNIX-based systems compliant with the POSIX [14] standard. Programming directly with such real-time operating system APIs—without explicit support from a programming language—is both complex and error prone: large code sections need to be written to setup timers, to handle different time formats, and to correctly implement safe synchronization mechanisms for communication of data between tasks. Moreover, if a specific target API is chosen for an application, the program is no longer portable: all timing operations become platform specific.

In this paper, we propose a small set of language primitives that we use for creating an extension to the C programming language, called *Timed C*. The key idea is to insert *timing points* within the program, where the user can explicitly specify *soft* and *firm* real-time constraints. The two main design objectives are: i) *simplicity:* few expressive constructs can express arbitrary complex timing semantics, and ii) *portability:* the compiler can directly compile programs targeted for different RTOSs and hardware platforms. The Timed C language can be used directly as a programming language for real-time programming, or as a target language for code generation. In summary, we make the following contributions:

- We propose an extension to C, called Timed C, using a minimal number of language primitives. This includes timeliness primitives to program soft and firm tasks, concurrent primitives, and scheduling primitives. We show systematically how this set of primitives can be used to construct various timing behaviors. (Section III)

- We design and implement a retargetable *source-to-source compiler*[1] for Timed C. The current compiler can target POSIX compliant operating systems and FreeRTOS. (Section IV)
- We design and implement parts of the on-board computer software for an ongoing student satellite project using Timed C. We demonstrate and evaluate portability by compiling this application to two different hardware platforms. (Section V)

II. State Of The Art and Motivation

The concept of introducing timing primitives to a programming language is not new, and over the years many languages with timing primitives have been developed. In this section, we discuss the state-of-the-art languages for programming real-time systems and motivate the rationale for designing new timing primitives for the C language.

Real-time systems are intrinsically concurrent with temporal features. The main temporal requirements are: i) criticality of deadlines: soft, firm, and hard, ii) release intervals: periodic, aperiodic, and sporadic, and iii) access to clocks. The main concurrent aspects are: i) tasks/threads, ii) priorities, and iii) scheduling policy. In this section, the expressiveness of a programming language is discussed based on its ability to program these temporal and concurrent features. Along with expressiveness we also explore simplicity and portability. We focus on Ada, RTJS, and C-based languages and APIs because they are extensively used in the development of real-time systems. We conclude this section by providing motivation for extending the C programming language for programming real-time system.

A. Ada and Real-Time Java

Ada [15] was originally designed and developed for programming real-time systems [16]. Hence, Ada has many *direct* primitives for programming various temporal and concurrent requirements. Ada's delay primitives, `delay` and `delay until` enable the execution of a task to be delayed for a relative and an absolute time, respectively. The `delay until` primitive guarantees only a lower bound on the delay and does not prevent overruns. Instead, a periodic task with firm deadline is programmed using a combination of Ada's `delay until`, and `select-abort` statements. Since Ada does not have direct constructs for programming aperiodic and sporadic tasks, these are programmed using Ada's low-level primitives for interrupts. Concurrency is programmed using Ada's `task` construct. The scheduling policy and priority of a task is programmed using Ada's `Task_Dispatching_Policy` and `Priority` pragmas, respectively. See Appendix VIII-A for an Ada example of a periodic task with firm deadline.

The Real-Time Java Specification (RTJS) [8], [17] also introduces several *direct* primitives for programming real-time systems. The concurrent unit of execution in RTJS is either an instance of class `RealtimeThread` or a class extending `RealtimeThread`. A periodic thread is specified using `PeriodicParameters`. The `waitForNextPeriod` programs the delay and unblocks the thread at the start of each release. Unlike Ada, RTJS has direct constructs for expressing aperiodic and sporadic threads, which are programmed using instances of `AperiodicParameters` and `SporadicParameters`, respectively. A thread with firm deadline in RTJS is programmed using an instance of the `Timed` class that interrupts the thread on timeouts. The scheduling policy is declared using an instance of `Scheduler`, together with the `setScheduler` method, and priority using the method `setPriority`. An example code in RTJS for a periodic task with firm deadline is listed Appendix VIII-B.

Ada and RTJS are portable across several operating systems and hardware platforms [18], [19], and Ada has been one of the favoured languages in industry for the development of safety-critical systems. However, when it comes to the development of small-scale embedded systems using bare metal, the most common choice of programming language is C [20]. One of the reasons for C being the popular choice is because most of these small-scale embedded systems are developed on small microcontroller units and bare metal platforms, which are running lightweight RTOSs that support programming in C.

B. C Based Programming Languages and API

Unlike Ada and RTJS, ANSI C has no constructs for temporal and concurrent requirements. To program real-time systems, a combination of C and the POSIX standard [14], referred to as Real-Time POSIX C in this paper, is typically used. The POSIX standard introduces several APIs to interface across several UNIX-based operating systems. The concurrent unit of execution in Real-Time POSIX C is `pthread`. Real-Time POSIX C does not provide direct primitives for programming periodic, sporadic, and aperiodic tasks. These primitives are programmed using Real-Time POSIX APIs for delay, timer, and interrupt. The `clock_nanosleep` construct is Real-Time POSIX C counterpart of Ada's `delay until`. The scheduling policy and priority are programmed using `sched_setscheduler` and `setpriority`, respectively. An example code in Real-Time POSIX C for a periodic task with firm deadline is listed in Appendix VIII-E.

Although Real-Time POSIX C does not provide direct constructs for programming the various real-time aspects, its low-level APIs can be combined to achieve such tasks. In terms of expressiveness, Real-Time POSIX C is comparable to Ada and RTJS [20]. However, Real-Time POSIX C is more verbose. The full page program listed in Appendix VIII-E involves setting up timers, signals, and other user defined functions. As the complexity of real-time systems increase, the

[1]KTC stands for KTH's Timed C compiler. The complete compiler is available as open source: http://github.com/timed-c/ktc

size of the applications implemented using Real-Time POSIX C also increases. Apart from simplicity, an increasing code size has direct impact on the productivity and maintainability of the program.

Besides the Real-Time POSIX API, only a few C languages with temporal constructs exist. Real-Time Concurrent C [9] is a superset of Concurrent C (an extension of C for parallel programming) that provides primitives for delaying program, specifying periodicity, and deadlines. Real-Time Concurrent C was designed to execute on an UNIX-based implementation of Concurrent C. To the best of our knowledge, the compiler support for Real-Time Concurrent C is not available anymore. The Real-Time Concurrent C pseudo program listed in VIII-C implements a periodic loop with firm deadline. Although Real-Time POSIX C and Real-Time Concurrent C are portable to platforms running UNIX-based operating systems, some of the lightweight RTOSs for small-scale MCUs, such as FreeRTOS [13], Erika [21], and Contiki [22], are not POSIX compliant.

C. Other Related Work

Arduino is an open-source software and hardware platform for embedded computing. The Arduino programming language is based on C++ and consist of various functions and libraries. It is portable across all Arduino-compatible platforms. The timing functions of the standard Arduino API, `delay` and `delayMicroseconds`, enable the execution of a task to be delayed for a relative time. The code listed in Appendix VIII-F implements a periodic task with firm deadline. Since standard Arduino provides no support for absolute delay or timers the example in Appendix VIII-F uses an external timer library. The standard Arduino API supports only single-threaded applications. Qduino [23] is an extension to the Arduino API that provides support for multithreading along with communication and synchronization between the threads. Qduino is portable across all Arduino-compatible devices running the Quest real-time operating system [24].

Bui et al. [25] identify four ways of controlling time by providing an extension to an instruction set architecture (ISA) with temporal semantics. Zimmer et al. [26] introduce a set of timing instruction for FlexPRET, a fine-grained multithreaded processor for mixed-criticality system. These timing instructions are implemented in C as inline assembly for the RISC-V ISA. This line of work has inspired the development of our approach.

The Real-Time Euclid [27] programming language supports timing constructs that guarantees schedulabilty, and programming of reliable real-time systems. PEARL [28] is an industrial programming language that supports hardware independent programming of multi-task real-time applications. Modeling languages, such as UML MARTE [29], Simulink [30], Modelica [31], Ptolemy [32], and domain-specific languages embedded in Modelyze [33], all have temporal semantics. Models developed in some of these environments can be compiled into C code. The Giotto [5] system allows a programmer to write platform independent Giotto programs for time sensitive control applications. Synchronous programming languages, such as Esterel [2], Lustre [1], and Signal [3] contain logic timing primitives that are used for implementing safety-critical reactive systems. The programming language *nesC* [34] for networked embedded systems supports event-driven execution and a flexible concurrency model. Finally, co-simulation environments and standards [35] enable interoperability between different timed languages.

D. Motivation

In summary, along with Ada and RTJS, C is also a popular choice of programming language for developing real-time applications [20]. However, standard C lacks timing constructs, and programming with the Real-Time POSIX API results in large and error prone programs. Although programs using the POSIX API are portable across various UNIX-based operating systems, far from all real-time applications are developed on UNIX. As a solution to these problems, we introduce a programming language called Timed C. This language extends the C programming language by introducing constructs for timing, concurrency, and scheduling. To illustrate Timed C's design objective of simplicity, we compare the full page POSIX C code listed in Appendix VIII-E to the following Timed C code, which performs the exact same task.

```
void main(){
  while(1){
    sense();
    fdelay(30, ms);
  }
}
```

The exact meaning of this program will be discussed in the next section. Although comparing lines of code is seldom a fair comparison, a few lines of code is clearly favorable compared to a full page of code of setting up timers etc. To verify the expressiveness of Timed C, we have programmed various real-time aspects discussed in [20] using Ada, RTJS, Real-Time Concurrent C, and Timed C[2].

III. Timing Primitives in Timed C

In this section, we propose a set of language primitives for programming with time in C. These primitives are classified into five different types: i) *soft timing-point primitives*, ii) *absolute time primitives* iii) *firm timing-point primitives*, iv) *timing-point primitives for arbitrary deadline* v) *concurrent primitives*, and vi) *scheduling primitives*. A table summarizing all Timed C primitives is listed in Appendix VIII-G.

A. Soft Timing-Point Primitives

We use the concept of *timing points* to handle the various timing constraints. From a programmer's perspective, if there are no timing violations, time only elapses at timing points. The code that is executed between timing points should conceptually be seen to take zero time. However, if the timing constraints are violated, the different timing points enable the programmer to react in time. That is, as long as there are no

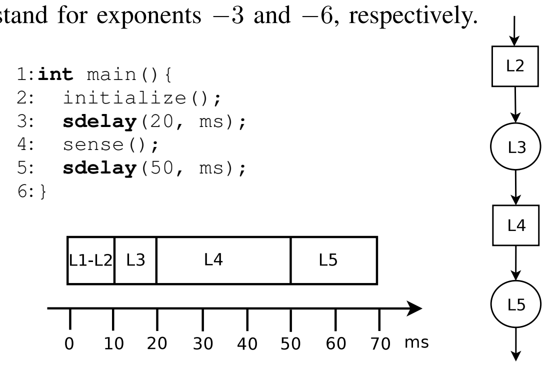
[2]This benchmark suite is available at : https://github.com/timed-c/ktc/tree/master/benchmark

timing violations, the *logical time* conforms to the *real time*. If a timing violation occurs, the programmer can explicitly react to the real-time behavior. In this language, the timing primitives `sdelay`, `stp`, `fdelay`, `ftp`, and `gettime` are timing points. The start of a function is implicitly considered to be a timing point.

In Timed C, a soft timing point is specified using the following statement

```
sdelay(expr, n)
```

where `expr` is either an integer literal or a C expression that evaluates to an integer. Argument `n` represents the resolution exponent, where the resolution is 10^n seconds. The user does not have to explicitly write the exponent. Instead, standard definitions, such that `ms` (milliseconds) and `us` (microseconds) stand for exponents -3 and -6, respectively.



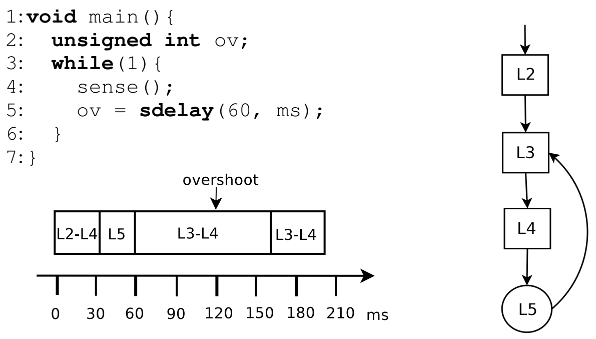
```
1:int main(){
2:  initialize();
3:  sdelay(20, ms);
4:  sense();
5:  sdelay(50, ms);
6:}
```

Fig. 1: A diagram depicting a simple function that uses two `sdelay` timing points. Note that time between the start of the function and the first `sdelay` (line 3) takes 20 ms, even if the execution of `initialize()` takes less than 20 ms. Similarly, the time between the first and second `sdelay` (line 5) is 50 ms.

Fig. 1 shows a function, its timing diagram, and its control flow graph (CFG). In the CFG, the circles and squares depict timing points and other programming constructs, respectively. The primitive `sdelay` specifies the relative delay from the previous timing point. For instance, when the function in Fig. 1 is executed, the `sdelay` at line 5 ensures that the total time taken to execute the code between the two `sdelay` statements at line 3 and 5 is *equal to or greater than* 50 ms. The soft timing point ensures a lower bound on the specified delay. If the time taken to execute the segment of code between two timing points is greater than the specified delay, the `sdelay` primitive returns the amount of overshoot as listed at line 5 in Fig. 2. If there is no overshoot, zero is returned.

Consider now Fig. 2 that shows a timed periodic loop. Assume that the function starts to execute line 1 at $t = 0$ and that the `sdelay` at line 5 completes its first iteration at $t = 60$. In the second iteration, assume that at $t = 120$ `sense` has still not finished executing. If the execution of `sense` finishes at $t = 160$, the `sdelay` at line 5 returns 40 ms as the overshoot. The timed loop in Fig. 2 can also be written as a C macro, as listed in Appendix VIII-D.

Note that the overshoot in Fig. 2 results in a scenario where the periodic loop becomes out of phase. In some applications,



```
1:void main(){
2:  unsigned int ov;
3:  while(1){
4:    sense();
5:    ov = sdelay(60, ms);
6:  }
7:}
```

Fig. 2: A function implementing a periodic loop using `sdelay`. The example shows an overshoot at time 160.

this might be the expected behavior, whereas sometimes the expected behavior is to keep the loop in phase. Fig. 3 shows example code where an overshoot is compensated with a skipped period, to become in phase again. This example demonstrates the first use of *reacting* to real time. Assume at $t = 160$ that the `sdelay` at line 5 returns 40 ms as the overshoot. Then, the guard of the `while` statement at line 6 evaluates to `true` and the `sdelay` at line 7 is executed. The `sdelay` then delays 20 ms since the execution of line 5 to compensate for the overshoot, and gets back in phase.

We need to give two important remarks for this example. Firstly, a careful reader might think that there can be a small drift if the argument to the `sdelay` at line 7 is close to zero. However, this cannot happen because it is compensated in the next period in the `while` loop. Secondly, a naive compilation of the example program would use relative delays, that is, to delay 60 ms from the previous timing point. This would introduce a small drift in every period. *Although relative time delays are used in the program (for convenience), the compiler will actually use absolute delays and timers internally.* As a consequence, the example will not introduce any drift during runtime and will always be in phase, on any supported platform.

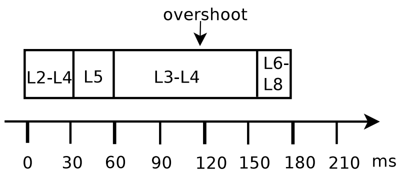
B. Absolute Time Primitives

The language primitive `gettime(n)` is a timing point that returns the time elapsed since epoch in the resolution of `n`. In many real-time systems, an operation needs to be executed at a specified absolute time. This can be programmed using a combination of `gettime` and `sdelay`. Fig. 4 shows a code where `actuate` is executed at an absolute time specified by `actuateAtTime`. Assume that `tcomp` is set to 1506968000 (time value of October 2, 2017 18:13:20 since epoch). The `gettime` at line 4 sets `tnow` to 1506967980 (October 2, 2017 18:13:00). Then the `sdelay` at line 5 delays 20 seconds, and `actuate` starts executing at 1506968000. Because the `sdelay` statement implements a delay relative to the previous timing point (`gettime` in this case), the absolute delay becomes exact, even if a relative delay construct is used.

C. Firm Timing-Point Primitives

A computation with firm timing requirement loses its utility on not meeting its deadline. In such scenario, the execution of

```
 1:void main(){
 2:  unsigned int ov;
 3:  while(1){
 4:    sense();
 5:    ov = sdelay(60, ms);
 6:    while(ov > 0){
 7:      ov=sdelay(60-ov%60, ms);
 8:    }
 9:  }
10:}
```



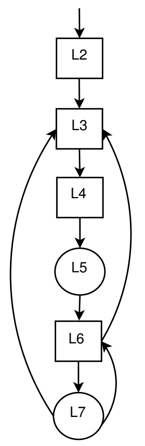
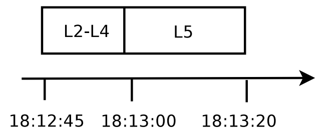


Fig. 3: A function implementing a periodic loop using `sdelay`, illustrating an error handling mechanism that ensures that the overshoot is compensated to make it stay in phase.

```
1:int main(){
2:  long tcomp, tnow;
3:  tcomp = actuateAtTime();
4:  tnow = gettime(sec);
5:  sdelay(tcomp - tnow, sec);
6:  actuate();
7:}
```



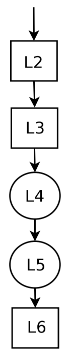


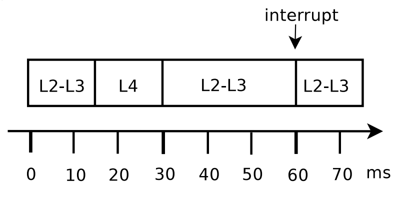
Fig. 4: A function implementing a delay until a specified absolute time using `gettime`.

this time critical computation should be aborted. We introduce firm timing points to handle this type of timing requirement. Firm timing points are specified using the keyword `fdelay`, as shown in the following statement:

```
fdelay(expr, n)
```

Similar to the behavior of `sdelay`, `fdelay` also introduces a delay relative to the previous timing point. In addition to enforcing a lower bound, `fdelay` also ensures an upper bound. For example, in Fig. 5, suppose `main` starts at $t = 0$. In the first iteration of the `while` loop, suppose `sense`

```
1:void main(){
2:  while(1){
3:    sense();
4:    fdelay(30, ms);
5:  }
6:}
```



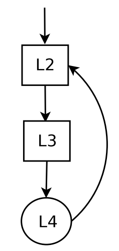
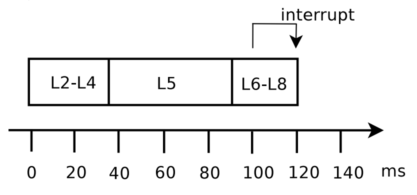


Fig. 5: A simple periodic loop using the `fdelay` construct.

completes at $t = 15$. Then `fdelay` delays until $t = 30$. However, in the second iteration at $t = 60$, `fdelay` interrupts the execution of function `sense`, and passes the control to the next iteration directly after the declared `fdelay` statement.

In some cases, interrupting a computation may lead to undesirable or incomplete results. In order to ensure that such computations are not interrupted by an `fdelay`, we introduce the language primitive `critical`. We illustrate the use of `critical` in Fig. 6. Anytime algorithms are a class of algorithms that initially compute a suboptimal solution and as time passes, the quality of the solution is improved [36]. The language primitives `fdelay` and `critical` can be used to implement an anytime algorithm, as illustrated in Fig. 6. The function `computePath` is assumed to compute a path for navigation. A feasible, but suboptimal path is computed by `initialize`. Function `computeAnytime` is assumed to improve the result, by reading from `a` and writing to `b`. Now, suppose `computePath` started at $t = 0$ and `computeAnytime` completes at $t = 90$. If `memcpy` (copying from `b` to `a`) is still executing at $t = 100$, then the interrupt is delayed until the execution exits from the critical section. The figure shows a 20 ms delay because of the critical section, but the actual timing is by design application dependent.

```
 1:void computePath(int* a){
 2:  int b[100];
 3:  initialize(a);
 4:  while(1){
 5:    computeAnytime(b, a);
 6:    critical{
 7:      memcpy(a, b, 100);
 8:    }
 9:  }
10:  fdelay(100, ms);
11:}
```



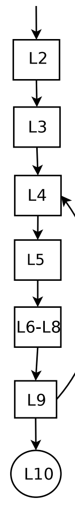


Fig. 6: A function implementing an anytime algorithm by using `fdelay` and `critical`.

In Timed C, nested timing points can be programmed using function calls. In the example below, `compute` imposes an inner timing constraint within the outer timing constraint of `main`. Here, if `compute` is still executing 50 ms after its start, the `fdelay` at line 8 interrupts its execution.

```
1:void compute(){
2:  control();
3:  sdelay(30, ms);
4:}
5:void main(){
6:  int a;
7:  compute();
8:  fdelay(50, ms);
9:}
```

D. Timing-Point Primitives for Arbitrary Deadline

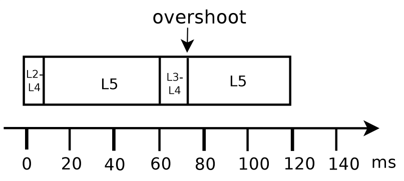
In order to support soft and firm timing constraints with arbitrary deadlines, we introduce the language constructs `stp` (soft timing point) and `ftp` (firm timing point), respectively.

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

Argument `expr1` is the lower timing bound (the amount of time to delay) and `expr2` is the upper bound (the deadline). Note that the timing primitive `sdelay` is actually a special case of `stp`, where the upper and lower bounds are equal. Similarly, if the upper and lower bounds are equal, `fdelay` can be used instead of `ftp`.

Consider Fig. 7 that shows a timed periodic loop with a firm timing constraint. Assume that the function starts to execute at $t = 0$ and `sense` at line 4 completes at $t = 5$. The `ftp` at line 5 delays the execution until $t = 60$ (lower bound). In the next iteration, assume that `sense` at line 4 is still executing at $t = 70$. The execution of `sense` is then interrupted at $t = 70$ (because of the upper bound 10 ms). Note that the `ftp` construct at line 5 does not finish until $t = 120$, because of the lower bound delay of 60 ms.

```
1:void main(){
2:  unsigned int ov;
3:  while(1){
4:    sense();
5:    ftp(60, 10, ms);
6:  }
7:}
```



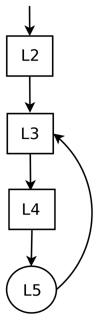


Fig. 7: A function implementing a periodic loop using `ftp`.

E. Concurrent Primitives

The construct `task` provides support for concurrent asynchronous tasks. A function declared as `task` is instantiated as a task when it is called from another function. In Fig. 8, the tasks `bar` and `foo` are instantiated in `main`.

Communication between concurrent tasks is supported using two different types of channels: the *FIFO channel* and the *Latest Value (LV) channel*. FIFO and LV channels are declared using the primitives `fifochannel` and `lvchannel`, respectively. In `fifochannel`, the read is a blocking operation, whereas write is non-blocking. This can be used to synchronize concurrently running asynchronous tasks. In contrast to FIFO channels, the `lvchannel` supports a wait-free mechanism for read and write operations. Both the read and write operations are non-blocking. Here, the latest value written to the channel is the one available to the reader. The `lvchannel` can be used to implement real-time systems where the control algorithm is executed using the latest sensor value [37]. The `lvchannel` supports multiple readers and single writer. The `fifochannel` supports single reader

```
int lvchannel chan1;
task bar(){
  int c;
  while(1){
    c = sense();
    cwrite(chan1, c);
  }
}
task foo(){
  int d;
  while(1){
    cread(chan1, d);
    compute(d);
    sdelay(60, ms);
  }
}
void main(){
  bar();
  foo();
}
```

Fig. 8: A program implementing two tasks that communicate using an `lvchannel`.

and single writer. The primitive `multilvchannel` extends `lvchannel` to support multiple readers/writers. Similarly, `multififochannel` extends `fifochannel` to multiple readers/writers.

The read and write operations for both these channels are performed using the primitives `cread` and `cwrite`:

```
cread(chn, data)
cwrite(chn, data)
```

In the examples above, `chn` specifies the channel and `data` is the variable that should be written to (in case of a `cwrite`), or read from (in case of a `cread`). Both `cwrite` and `cread` are atomic. In Fig. 8, `chan1` is an LV channel with a single reader, `foo`, and a single writer, `bar`. Note, when `cread` returns, `d` contains the value read from `chan1`. A `fifochannel` can be used in a similar way.

F. Scheduling Primitives

In Timed C, the scheduling policy of a task is set using the following statement

```
spolicy(policy)
```

where `policy` specifies the scheduling policy. In Rate Monotonic (RM) and Deadline Monotonic (DM) scheduling, the priority of a task depends on its period and deadline, respectively. In the Earliest Deadline First (EDF) scheduling scheme, the priority of a task depends on its runtime, deadline, and period. In applications using RM and DM scheduling, the priorities of the tasks are usually calculated offline and specified in the program by the programmer. Programming languages, such as Ada and Real-Time POSIX C, have special constructs for specifying priority, and EDF scheduling parameters. However, the priority of a task in all these scheduling schemes is a consequence of its timing constraint, and in Timed C these timing constraint are specified using timing primitives. A unique property of Timed C is that priorities are determined and set implicitly by performing static analysis on the timing

```
 1:task foo(){
 2:  spolicy(EDF);
 3:  while(1){
 4:    senseA();
 5:    sdelay(30, ms);
 6:  }
 7:}
 8:task bar(){
 9:  spolicy(EDF);
10:  while(1){
11:    senseB();
12:    sdelay(50, ms);
13:    sdelay(15, ms);
14:  }
15:}
16:void main(){
17:  initialize();
18:  foo();
19:  bar();
20:}
```

Fig. 9: A program implementing two tasks with EDF scheduling policy.

points. Fig. 9 shows two tasks scheduled with EDF scheduling policy, specified as EDF in line 2 and line 9.

Depending on the underlying operating system, the `spolicy` construct in Timed C can support EDF, RM with First-In, First-Out (FIFO), RM with Round-Robin (RR), DM with FIFO, and DM with RR scheduling policies specified as EDF, `FIFO_RM`, RR_RM, `FIFO_DM`, and RR_DM, respectively.

The varying timing constraints of an aperiodic task can be programmed using Timed C's timing primitive, where the delay is specified as a C expression. In order to infer the priorities of such tasks, we introduce the construct

```
aperiodic(value, n)
```

where `value` is an integer literal specifying the period and `n` is the resolution exponent. As for future work, we plan to integrate various aperiodic scheduling algorithms [38] and input/output handling, to better support scheduling of aperiodic tasks. For sporadic tasks, the inter-arrival time of a task can be smaller than the time taken to execute the task, resulting in burst behavior. This can be programmed using a combination of the `sdelay` and the `task` constructs[3].

Note also that the default implicit mechanism for determining priorities can be overridden using the following statement

```
spriority(priority)
```

where `priority` specifies the task's priority as an integer value. If the `spolicy` construct is not specified, the default policy for the specific platform will be used.

IV. Implementation

KTC is a source-to-source compiler that compiles a Timed C file into a target specific C file. It also performs static analysis and rejects programs with incorrect timing behaviour. In its current version, the compiler supports compilation into POSIX.4 and FreeRTOS C files. Note that the existing weaknesses of the C programming language are not handled by the Timed C compiler, as this is not the focus of this paper. Future work may include the integration of our compiler with existing static analyzers and bug detection tools.

The various steps involved in compiling a Timed C file in our source-to-source compiler is depicted in Fig 10. The compilation is divided into two parts: a front-end and a back-end. The different phases *CIL frontend*, *initial analysis*, *static analysis*, *transformation*, and *CIL code generation* are represented as boxes. The various actions performed by the different phases are depicted within these boxes.

A. KTC Front-End

The front-end parses the input file, generates an *abstract syntax tree (AST)* and performs static analysis.

CIL Front-End: KTC is built on top of the CIL (C Intermediate Language) framework [39]. We add the timing constructs to the CIL front-end, which parses a Timed C file, and generates a CIL AST.

Initial Analysis: The timing requirement of `fdelay` is implemented using timers and labels. Hence, unique labels are assigned to the firm timing points in the CIL AST. Since both the `lvchannel` and the `fifochannel` use the same set of constructs for read and write, a hash table is created that maps the various channels to its type and its number of readers and writers. This information is used in the static analysis and transformation phases discussed later in this section. Functions declared as `task` are added to a list and this list is used to instantiated these functions as tasks.

Static Analysis: The first part of the static analysis is to create firm successors of timing points. In the static analysis phase, programs with incorrect timing behaviours are rejected. For instance, in the following example, only one of the `fdelay` statements (line 5 or line 9) will be executed. To implement the correct timing behaviour of the this program, we need to know which branch will be taken. Since this information cannot be safely determined at compile time, such programs are rejected.

```
 1:void main(){
 2:  int x;
 3:  x = foo();
 4:  if(x){
 5:    computeA();
 6:    fdelay(10, us);
 7:  }
 8:  else{
 9:    computeB();
10:    fdelay(50, us);
11:  }
12:}
```

B. KTC Back-End

The back-end transforms the language primitives and emits a platform dependent C file.

Transformation: The lower bound of a timing point is enforced using a function that delays the execution until an absolute time. The delay is based on absolute time to avoid any timing *drift* in the program. The timing constraint of `fdelay` is implemented using absolute timers, `setjmp`

[3]See file `ioburst.c` on GitHub in folder `rtas/examples/`

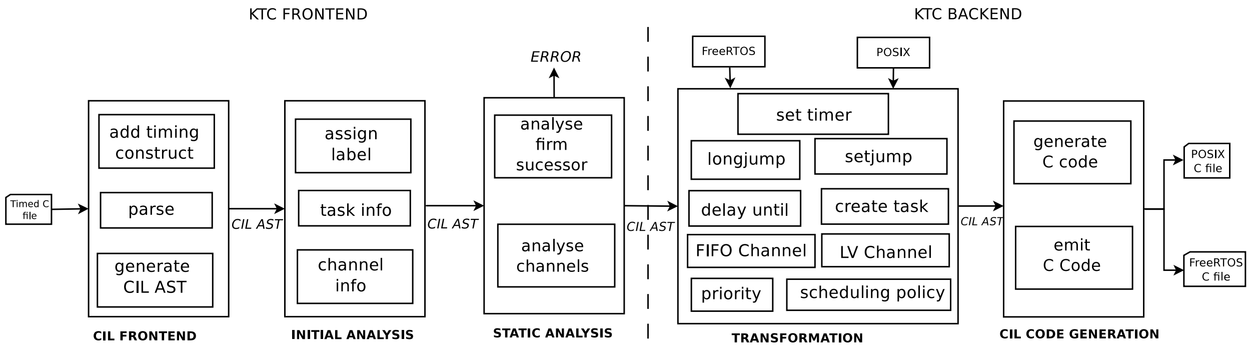


Fig. 10: The different phases of the KTC source-to-source compiler.

and `longjmp`. Calls to functions (declared as `task`) are transformed to create tasks using the list from the initial analysis. A call to `spolicy` is transformed to set scheduling policy. A FIFO queue of the specified data type is used to implement a `fifochannel`. Depending on the number of readers, either the Simpson's four-slot algorithm [40] or the Cyclic Asynchronous Buffers mechanism [41] is implemented for a `lvchannel`.

In general, the specific transformation depends on the type of target platform. For example, `sdelay` is implemented using `clock_nanosleep` for POSIX.4 and `vTaskDelayUntil` for FreeRTOS. For the construct `task`, we use *pthreads* for POSIX.4, and *FreeRTOS tasks* for FreeRTOS. To assign priorities, we use `vTaskPrioritySet` for FreeRTOS and `sched_setattr` for POSIX.4.

CIL Code Generation : In the final phase, the CIL framework emits the POSIX.4 or the FreeRTOS C file.

V. Case Study: On-board Satellite Computer

To evaluate the portability and simplicity of using Timed C to program real-time systems, we implemented the timing parts for an on-board satellite computer system. More specifically, the case study is part of the MIST (MIniature Student saTellite) project, an ongoing CubeSat student project at the KTH Royal Institute of Technology in Stockholm. The different components and subsystems of the satellite, discussed in Section V-A, are being developed by different sub-teams at different locations. In the final stage of the satellite development, the various components will be integrated and tested. The *on-board computer (OBC)* on the satellite plays a central role as it is responsible for data handling and communication. Access to the OBC (the actual flight hardware) during the development phase is important to the different sub-teams of the project. Moreover, having an OBC to test the various functionalities during development can also speed up the final integration and testing of the satellite. However, the flight qualified OBC is very expensive hardware. An alternative is to compile the OBC software to cheaper platforms that can be used during testing and development. However, for such an approach to be feasible, both the logical and temporal behavior must be portable and repeatable when compiling to different hardware platform.

In this case study, we implement the OBC software, discussed in Section V-B, using Timed C. We then evaluate if the same implementation can be compiled both to a less expensive platform and to the real flight hardware, and still get the same temporal and logical behavior.

A. System Architecture of MIST

MIST is a scientific research satellite with seven experimental payloads. It is a low earth orbit (LEO) satellite following the sun-synchronous orbit. The basic operations of the MIST satellite are to orbit the earth, control the experiments, collect payload data from the experiments, collect housekeeping data (thermal and power readings), transmit housekeeping and payload data to the ground station, and receive and execute commands from the ground station. The data transmitted by the satellite to the ground station is referred to as *telemetry*, and the commands sent by the ground station to the satellite are called *telecommands* (TCs). Since MIST is a LEO satellite having intermittent contact with the ground station, the design goal is to store and execute telecommands at some future time. These are referred to as *time tagged* telecommands. The main subsystems of the MIST satellite are: on-board computer (OBC), radio, attitude control system, electrical power system, and the ground system.

B. Software Architecture of OBC

The OBC is responsible for performing several functions concurrently. Some of these functions, such as the attitude control and collecting housekeeping data, are performed periodically, whereas the execution of telecommands are event driven. Hence, we designed the OBC software as a set of periodic and aperiodic real-time tasks as depicted in Fig. 11. Channel `tcChan` is a `fifochannel`, and hence if it is empty `executeTC` blocks waiting for `retrieve` to write

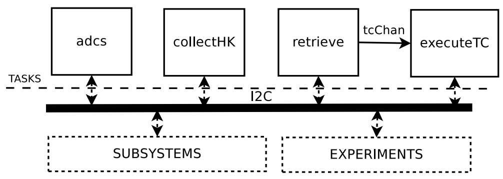


Fig. 11: The software architecture of the MIST satellite.

```
task executeTC(){
  struct_tcmist tc, tcnext;
  struct_tcmist* tctop = NULL;
  long t = 0, tctime = 0;
  spolicy(FIFO_RM);
  aperiodic(4000, ms);
  while(1){
    printf("executeTC:");
    printTrace();
    tctop = top(prioq);
    if(tctop == NULL)
      tctime = INFINITY;
    else
      tctime = tctop->aTime;
    tcnext.rx_length = 0;
    t = gettime(sec);
    cread(tcChan, tcnext);
    ftp(0, tctime-t, sec);
    if(tcnext.rx_length != 0){
      push(prioq, &tcnext);
    }
    tctop = top(prioq);
    t = gettime(sec);
    if(t >= tctop->aTime){
      pop(prioq, &tc);
      executeTCAux(tc);
    }
  }
}
```
(a)

```
task executeTC(){
  struct_tcmist tc;
  struct_tcmist* tctop = NULL;
  long t = 0, tctime = 0;
  spolicy(FIFO_RM);
  aperiodic(4000, ms);
  while(1){
    printf("executeTC:");
    printTrace();
    if(nelem(&tcChan) != 0
      || prioq->len == 0 ){
      cread(tcChan, tc);
      push(prioq, &tc);
    }
    tctop = top(prioq);
    tctime = tctop->aTime;
    t = gettime(sec);
    sdelay(tctime - t, sec);
    pop(prioq, &tc);
    executeTCAux(tc);
  }
}
```
(b)

```
task executeTC(){
  struct_tcmist tc;
  struct_tcmist* tctop = NULL;
  long t, tctime = 0;
  long threstime = 4;
  int num;
  spolicy(FIFO_RM);
  aperiodic(4000, ms);
  while(1){
    printf("ExecuteTC:");
    printTrace();
    if(nelem(&tcChan) != 0
      || (prioq->len == 0)){
      cread(tcChan, tc);
      push(prioq, &tc);
    }
    tctop = top(prioq);
    tctime = tctop->aTime;
    t = gettime(sec);
    if(threstime < (tctime -t))
      sdelay(threstime, sec);
    else
      sdelay(tctime - t, sec);
    if(t >= tctop->aTime){
      pop(prioq, &tc);
      executeTCAux(tc);
    }
  }
}
```
(c)

Fig. 12: Code listing (a) the immediate execution of telecommands using `ftp`, (b) the delayed execution of telecommands using `sdelay`, and (c) the execution of telecommands with bounded delay using `sdelay`.

to it. The priority queue, accessed by `executeTC`, stores telecommands in increasing order of their time tags.

The functions executed by the `adcs`, `collectHK`, and `retrieve` are periodic in nature. Hence, these are programmed as periodic tasks. On the other hand, the period of the task `executeTC` depends on the time tag of the telecommands and is not periodic. Based on the system requirement of the satellite and the criticality of an operation, the `adcs`, `retrieve`, and `collectHK` tasks are assigned periods of 1s, 3s, and 4s, respectively. The code for `retrieve` is listed below:

```
task retrieve(){
  struct_tcmist tc;
  spolicy(FIFO_RM);
  while(1){
    /* Code to retrieve and
         store telecommands  */
    if(tc.rx_length != 0){
      cwrite(tcChan, tc);
    }
    sdelay(3, sec);
  }
}
```

We implemented three different designs for the aperiodic task `executeTC`, all listed in Fig. 12. The ideal implementation is given in Fig. 12(a). If `prioq` is empty, `executeTC` blocks until a telecommand arrives on `tcChan`. Otherwise, `executeTC` delays its execution until it is time to execute the telecommand at the head of `prioq`. During this time if a new telecommand arrives on `tcChan` then `executeTC` processes this new telecommand. This design aims at immediately executing every telecommand received on `tcChan`. This design combines `ftp`, and `cread` to both wait for scheduled telecommands, and to interrupt the delay if a more urgent telecommand arrives. However, as it turns out, the MIST OBC flight hardware does not support timers, which makes it impossible to compile the `ftp` construct.

An alternative is shown in Fig. 12(b), which is using `sdelay` instead of `ftp`. As an unfortunate consequence, urgent telecommands cannot interrupt the scheduled telecommands. When `prioq` is empty, `executeTC` blocks on `tcChan`. Otherwise, it delays its execution until it is time to execute the telecommand at the head of `prioq`. All new telecommands received on `tcChan` during this time are not processed directly. The processing of a telecommand with a large time tag will delay the execution of all new telecommands. This increases the overall response time of the telecommands. Here we use `sdelay` instead of `stp` because the system has an implicit deadline.

Finally, Fig. 12(c) shows an approach using `sdelay`, but with bounded response time. The key idea for the latter approach is to never wait longer than a specified threshold time, in this case 4s. In all three implementations, the task is defined as aperiodic using the `aperiodic` construct.

C. Evaluation and Results

The portability of the Timed C programming language is evaluated by compiling the OBC software into two different platforms: i) A space qualified OBC hardware with a 400 MHz ARM 9 processor running FreeRTOS operating system with cooperative multitasking, and ii) Raspberry Pi 2 Model B with a 900 MHz ARM Cortex A7 CPU running Raspbian patched with RT-Preempt.

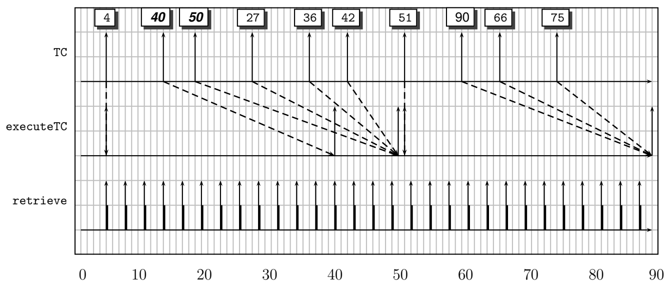


Fig. 13: Executing code in Fig. 12(b) on the OBC.

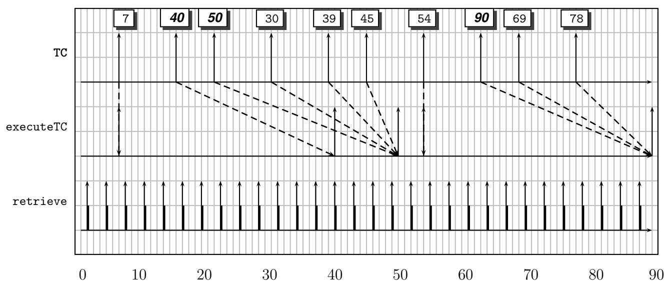


Fig. 14: Executing Fig. 12(b) on the Raspberry Pi.

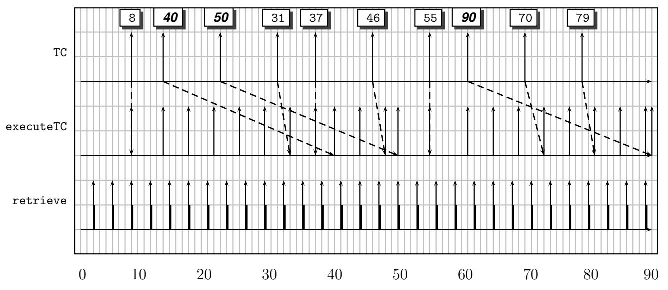


Fig. 15: Executing code in Fig. 12(c) on the OBC.

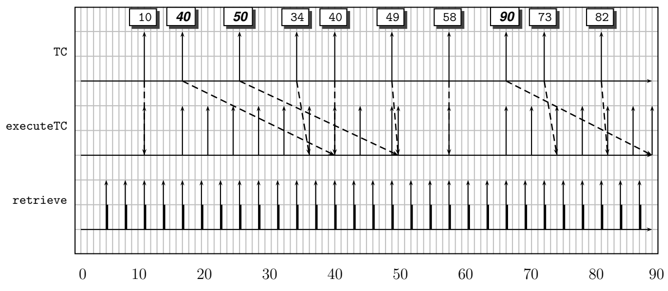


Fig. 16: Executing Fig. 12(c) on the Raspberry Pi

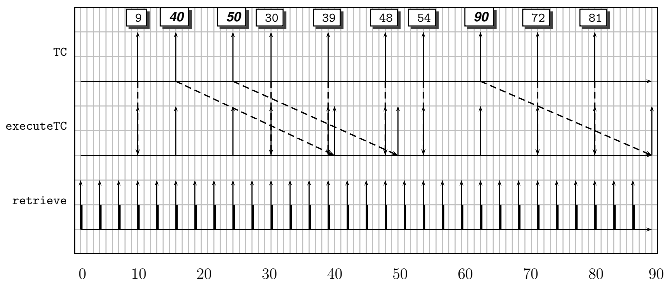


Fig. 17: Executing Fig. 12(a) on the Raspberry Pi.

Elveti [42], a flight proven mission control system (MCS) for small and nano satellites, is used to generate telecommands. MIST is an ongoing project, and many of its subsystems are still under development. Due to the unavailability of the radio, a radio simulator implemented on an Arduino Due was used to route the telecommands to the two hardware platforms. For the same reason, the `adcs`, and `collectHK` are skeleton tasks executing dummy code. We also consider a scenario, where there are no deadline misses. We evaluate the timing correctness of the task `executeTC` by sending random sequence of telecommands. Some commands are executed immediately, while others are time tagged for future execution.

Fig. 13 and Fig. 14 depict the execution traces of the OBC and the Raspberry Pi hardware, for the implementations listed in Fig. 12(b). An up arrow represents the release of a task. The row with label *TC* shows the arrival of telecommands. The numbers on the top represent the time tag, τ, of these telecommands. The time of execution of a telecommand time tagged for future execution is represented in bold, whereas the other commands are urgent. Due to the radio and the MCS, these commands arrive at slightly different times for the different platforms. The dotted line from *TC* to *executeTC* represents the instance of `executeTC` executing the telecommand. The τ of a telecommand to be executed immediately is equal to the absolute time at which it is received. In Fig. 13, a telecommand received at $t = 4$ is executed immediately. The next telecommand received at $t = 13$ with $\tau = 40$ suspends the execution of `executeTC` until $t = 40$. This stalls the execution of all immediate telecommands received between $t = 13$ to $t = 40$ increasing their response time. At $t = 40$, `executeTC` executes the telecommand with $\tau = 40$, and is suspended until $t = 50$. Note that it is only at $t = 50$ that all telecommands received between time 13 and 50 are executed.

Fig. 15 and Fig. 16 show the improved versions with bounded response time, according to the code in Fig. 12(c). Note, for instance at time 34 in Fig. 16, how an immediate command is not delayed until time 50, but executed almost immediately, within the 4s bounded response time.

Finally, recall the original efficient solution in Fig. 12(a), using `ftp`. The execution trace of this implementation on the Raspberry Pi is depicted in Fig. 17. Note again that this design is not possible to execute on the OBC flight hardware. The task `executeTC` is executed every time a telecommand arrives on `tcChan`, that is, there is no delay. For instance, at time $t = 30$ and $t = 39$, the commands are executed immediately.

VI. Conclusions

In this paper, we present Timed C, an extension to the C programming language with a minimal set of constructs for programming various aspects of real-time systems. We design and implement a source-to-source Timed C compiler, and conduct a case study for a real on-board satellite computer. As future work, we plan to integrate a WCET tool chain [43] into the Timed C compiler to provide hard timing constraints.

VII. Acknowledgment

The authors would like to thank Sven Grahn, Rodothea Myrsini Tsoupidi, Daniel Lundén, Christian Schulte, Johan Montelius, and Johan Engberg for providing valuable input throughout writing this paper. We also thank Gabriel Parmer, Björn Brandenburg, Edward Lee, and the anonymous reviewers for their insightful comments and suggestions. This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032).

References

[1] 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.

[2] 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.

[3] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[4] J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed real-time software for cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 45–59, 2012.

[5] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded software*. Springer, 2001, pp. 166–184.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 36:1–36:53, May 2008.

[7] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[8] A. Wellings, *Concurrent and real-time programming in Java*. John Wiley & Sons, 2004.

[9] N. Gehani and K. Ramamritham, "Real-time concurrent C: A language for programming dynamic real-time systems," *Real-Time Systems*, vol. 3, no. 4, pp. 377–405, 1991.

[10] V. F. Wolfe, S. Davidson, and I. Lee, "RTC: Language support for real-time concurrency," *Real-Time Systems*, vol. 5, no. 1, pp. 63–87, 1993.

[11] L. Palopoli, G. Buttazzo, and P. Ancilotti, "A C language extension for programming real-time applications," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE, 1999, pp. 103–110.

[12] Esterel Technologies, "SCADE Suite - Control Software Design Esterel Technologies," http://www.esterel-technologies.com/products/scade-suite/ [Last accessed: May 1, 2017].

[13] FreeRTOS, "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions," http://www.freertos.org [Last accessed: May 1, 2017].

[14] B. Gallmeister, *POSIX. 4 Programmers Guide: Programming for the real world.* " O'Reilly Media, Inc.", 1995.

[15] A. Wellings and A. Burns, "Real-time utilities for Ada 2005," in *Reliable Software Technologies–Ada Europe 2007*. Springer, 2007, pp. 1–14.

[16] P. A. Laplante and S. J. Ovaska, *Real-time systems design and analysis: tools for the practitioner*. John Wiley and Sons, 2011.

[17] E. J. Bruno and G. Bollella, *Real-Time Java Programming: With Java RTS*. Pearson Education, 2009.

[18] AdaCore., "Embedded Development — GNAT Pro — AdaCore," http://www.adacore.com/gnatpro/embedded/ [Last accessed: Sept. 23, 2017].

[19] F. Pizlo, L. Ziarek, and J. Vitek, "Real time Java on resource-constrained platforms with Fiji VM," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, 2009, pp. 110–119.

[20] A. Burns and A. J. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[21] "ERIKA Enterprise — Open Source RTOS OSEK/VDX Kernel," http://erika.tuxfamily.org/drupal/ [Last accessed: September 25, 2017].

[22] "Contiki: The Open Source Operating System for the Internet of Things," http://www.contiki-os.org [Last accessed: September 25, 2017].

[23] Z. Cheng, Y. Li, and R. West, "Qduino: A multithreaded arduino system for embedded computing," in *Real-Time Systems Symposium, 2015 IEEE*. IEEE, 2015, pp. 261–272.

[24] "Quest operating system," http://www.questos.org, accessed: 2018-02-06.

[25] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 274–279.

[26] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*. IEEE, 2014, pp. 101–110.

[27] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *Software Engineering, IEEE Transactions on*, no. 9, pp. 941–949, 1986.

[28] T. Mirtin, "Realtime programming language PEARL-Concept and characteristics," in *Computer Software and Applications Conference, 1978. COMPSAC'78. The IEEE Computer Society's Second International*. IEEE, 1978, pp. 301–306.

[29] "MARTE," http://www.omg.org/spec/MARTE/, accessed: 2017-09-25.

[30] "SIMULINK," http://www.mathworks.com/products/simulink/, accessed: 2017-04-19.

[31] "Modelica and the Modelica Association Modelica Association," https://www.modelica.org, accessed: 2017-09-25.

[32] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[33] D. Broman and J. G. Siek, "Gradually Typed Symbolic Expressions," in *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '18. New York, NY, USA: ACM, 2018, pp. 15–29.

[34] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *Acm Sigplan Notices*, vol. 49, no. 4, pp. 41–51, 2014.

[35] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate Composition of FMUs for Co-Simulation," in *Proceedings of the International Conference on Embedded Software (EMSOFT 2013)*. IEEE, 2013.

[36] S. Zilberstein and S. Russell, "Optimal composition of real-time systems," *Artificial Intelligence*, vol. 82, no. 1-2, pp. 181–213, 1996.

[37] G. Buttazzo, "Achieving scalability in real-time systems," *Computer*, vol. 39, no. 5, pp. 54–59, 2006.

[38] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[39] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL : Intermediate language and tools for analysis and transformation of C programs," in *International Conference on Compiler Construction*. Springer, 2002, pp. 213–228.

[40] H. Simpson, "Four-slot fully asynchronous communication mechanism," *IEE Proceedings E (Computers and Digital Techniques)*, vol. 137, no. 1, pp. 17–30, 1990.

[41] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[42] Elveti — Solenix GmbH, "Solenix.ch," https://www.solenix.ch/elveti [Last accessed: Oct. 06, 2017].

[43] D. Broman, "A Brief Overview of the KTA WCET Tool," *arXiv preprint arXiv:1712.05264*, 2017.

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

VIII. Appendix

In this appendix, we list examples of programming a periodic task with firm deadlines in Ada, RTJS, Arduino API, Real-Time Concurrent C, and Real-Time POSIX C. Note that these code listings are only partial. The complete implementations are available as part of the benchmark suit[4].

A. Ada

Fig. 18 lists an Ada program implementing a periodic loop with firm deadlines. The periodic release of the task is programmed using the `delay until` statement at line 21. The cumulative drift is eliminated at line 22. The firm timing requirement is implemented using the `select-abort` statement from line 16 to line 20. On a deadline overshoot, the execution of line 19 is aborted. The `Task_Dispatching_Policy` at line 1 and `Priority` pragma at line 8 set the scheduling policy and priority of the task, respectively.

```
 1:pragma Task_Dispatching_Policy(
      FIFO_Within_Priorities);
 2:with Ada.Text_Io; use Ada.Text_Io;
 3:with Ada.Real_Time; use Ada.Real_Time;
 4:with Ada.Real_Time.Timing_Events; use Ada.
      Real_Time.Timing_Events;
 5:with Example1; use Example1;
 6:procedure Firm is
 7:  task type Periodic_Firm is
 8:      pragma Priority(5);
 9:  end Periodic_Firm;
10:  task body Periodic_Firm is
11:     Next : Time;
12:     Interval : Time_Span := Milliseconds(30);
13:  begin
14:     Next := Clock + Interval;
15:     loop
16:      select
17:        delay until Next;
18:      then abort
19:        Sense; --read from sensor
20:      end select;
21:      delay until Next;
22:      Next := Next + Interval;
23:     end loop;
24:  end Periodic_Firm;
25:  ftask : Periodic_Firm;
26:begin
27:  Put_Line("Firm_Task!");
28:end Firm;
```

Fig. 18: An Ada program implementing a periodic loop with firm deadlines.

B. RTJS

Fig. 19 lists a Java program [44] that implements a periodic loop with firm deadlines. The period of the task `fd` is programmed using an instance of the class `PeriodicParameters` and the method `setReleaseParameters` at line 24 and line 26, respectively. The periodic release is programmed by the `waitForNextPeriod` statement at line 18. This statement eliminates cumulative drift. The firm timing requirement is implemented using an object of class `TimedOp` at line 15. On a deadline overshoot, the execution of the function at line 7 is interrupted.

```
 1:import javax.realtime.*;
 2:public class Firm extends RealtimeThread {
 3:  First obj = new First();
 4:  class TimedOp implements Interruptible {
 5:  public void run(
        AsynchronouslyInterruptedException ai)
 6:  throws AsynchronouslyInterruptedException {
 7:    obj.sense();//read from sensor
 8:  }
 9:  public void interruptAction(
10:    AsynchronouslyInterruptedException ai) {
11: }}
12: public void run(){
13:    RelativeTime intr = new RelativeTime(30, 0);
14:    Timed timed = new Timed(intr);
15:    TimedOp interuptible = new TimedOp();
16:    while(true){
17:      timed.doInterruptible(interuptible);
18:      waitForNextPeriod();
19:    }
20:  }
21:  public static void main(String[] args){
22:    Firm fd = new Firm();
23:    RelativeTime period = new RelativeTime(30,0);
24:    PeriodicParameters periodicParameters =
25:    new PeriodicParameters(null,period, null,null
         ,null, null);
26:    fd.setReleaseParameters(periodicParameters);
27:    fd.start();
28:  }
29:}
```

Fig. 19: A Java program implementing a periodic loop with firm deadlines.

C. Real-Time Concurrent C

The following code illustrates a Real-Time Concurrent C program implementing a periodic loop with firm deadlines.

```
1:void main(){
2:  while(1){
3:    every(30)
4:      sense();//read from sensor
5:  }
6:}
```

The periodic release of the task is programmed using `while` (line 2) and `every` (line 3). The firm timing requirement is implemented by `every` statement in line 3.

D. Examples of macros in Timed C

This example demonstrates how C macros can be used to construct new timed constructs by reusing Timed C primitives. The periodic release of a task is programmed using the macro `SOFT_PERIOD_LOOP` at line 5. The macro is defined on lines 1 and 2. The C preprocessor translates the code below to a code equivalent to the one listed in Fig. 2.

```
1:#define SOFT_PERIODIC_LOOP(expr, n, func)\
2:     while(1){func(); sdelay(expr, n);}
3:
4:void main(){
5:   SOFT_PERIODIC_LOOP(60, ms, sense);
6:}
```

[4]https://github.com/timed-c/ktc/tree/master/benchmark

E. Real-Time POSIX C

Fig. 20 lists a Real-Time POSIX program that implements a periodic loop with firm deadlines. The periodic release of the task is programmed using `clock_nanosleep` at line 48. The firm timing requirement is implemented using a timer, signal, `sigsetjmp`, and `siglongjmp`. On a deadline overshoot, the timer interrupt handler executes the callback function `timer_signal_handler`. Time in POSIX is represented as `struct timespec`. Note that the functions `convert_to_timespec` (line 19) and `add_timespec` (line 34) are user-define functions, defined outside this listing.

F. Arduino

The following code shows an Arduino program that implements a periodic loop with firm deadlines. The periodic release of the task is programmed using `loop` at line 14.

```
 1:#include <setjmp.h>
 2:#include "DueTimer.h"
 3:jmp_buf env;
 4:unsigned long tinit = 0;
 5:volatile int timer_interrupt = 0;
 6:void callback(){
 7:  timer_interrupt = 1;
 8:}
 9:void setup(){
10:  Timer3.setPeriod(30000);//30ms
11:  Timer3.attachInterrupt(callback);
12:  Timer3.start();
13:}
14:void loop() {
15:  int i=0;
16:  tinit = millis();
17:  i = setjmp(env);
18:  if(i == 0){
19:    sense();//read from sensor
20:  }
21:  Timer3.stop();
22:  timer_interrupt = 0;
23:  delay(30 - (millis() - tinit));
24:  Timer3.start();
25:}
```

G. Table Summarizing Timed C primitives

Timed C constructs	Functionality
`sdelay(expr, n)`, `stp(expr1, expr2, n)`	soft timing point
`fdelay(expr, n)`, `ftp(expr1, expr2, n)`	firm timing point
`gettime(n)`	returns the absolute time
`task`	creates a concurrent task
`lvchannel` `multilvchannel`	latest value channel
`fifochannel` `multififochannel`	FIFO channel
`cread(chn, data)`	read from channel
`cwrite(chn, data)`	write to channel
`spolicy(policy)`	specifies scheduling policy
`sprioity(priority)`	specifies priority as an integer
`aperiodic(value,n)`	period of an aperiodic task

```
 1:   /*Code using POSIX API*/
 2:int waiting_for_signal;
 3:jmp_buf env;
 4:void timer_signal_handler(int sig, siginfo_t*
        extra, void* cruft){
 5:  if(waiting_for_signal == 1){
 6:    siglongjmp(env, 3);
 7:  }
 8:  waiting_for_signal = 0;
 9:}
10:void main(){
11:  struct timespec start_time, interval_timespec;
12:  long interval;
13:  char* unit;
14:  int  ret_jmp;
15:  struct itimerspec i;
16:  struct sigaction sa;
17:  struct sigevent timer_event;
18:  timer_t mytimer;
19:  convert_to_timespec(&interval_timespec,3,"ms");
20:  sa.sa_flags = SA_SIGINFO;
21:  sa.sa_sigaction = timer_signal_handler;
22:  if(sigaction(SIGRTMIN, &sa, NULL) < 0){
23:    perror("sigaction");
24:    exit(0);
25:  }
26:  timer_event.sigev_notify = SIGEV_SIGNAL;
27:  timer_event.sigev_signo = SIGRTMIN;
28:  timer_event.sigev_value.sival_ptr=(void*)&
        mytimer;
29:  if(timer_create(CLOCK_REALTIME,&timer_event,&
        mytimer)<0){
30:    perror("timer_create");
31:    exit(0);
32:  }
33:  clock_gettime(CLOCK_REALTIME,&start_time);
34:  add_timespec(&(i.it_value ), start_time,
        interval_timespec);
35:  i.it_interval.tv_sec = 0;
36:  i.it_interval.tv_nsec = 0;
37:  if(timer_settime(mytimer, TIMER_ABSTIME, &i,
        NULL) < 0 ){
38:    perror("timer_setitimer");
39:    exit(0);
40:  }
41:  while(1){
42:    ret_jmp = sigsetjmp(env, 1);
43:    waiting_for_signal = 1;
44:    if(ret_jmp == 0){
45:      sense(); //read from sensor
46:    }
47:    waiting_for_signal = 0;
48:    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME
         ,&i.it_value, NULL);
49:    add_timespec(&(i.it_value ), i.it_value,
         interval_timespec);
50:    i.it_interval.tv_sec = 0;
51:    i.it_interval.tv_nsec = 0;
52:    timer_settime(mytimer, TIMER_ABSTIME, &i,
         NULL);
53:  }
54:}
```

Fig. 20: A Real-Time POSIX C program implementing a periodic loop with firm deadlines.