A Predictable Framework for Safety-Critical Embedded Systems

Sidharta Andalam ∗, Partha S Roop ∗, Alain Girault† and Claus Traulsen‡

∗ Department of Electrical and Computer Engineering, University of Auckland, New Zealand
Email: {sand080, p.roop}@aucklanduni.ac.nz
† INRIA Grenoble Rhône-Alpes, LIG, POP ART team, Grenoble, France
Email: Alain.Girault@inria.fr
‡ Christian-Albrechts Universitat zu Kiel, Germany
Email: ctr@informatik.uni-kiel.de

Abstract—Safety-critical embedded systems, commonly found in automotive, space, and health-care, are highly reactive and concurrent. Their most important characteristics are that they require both functional and timing correctness. C has been the language of choice for programming such systems. However, C lacks many features that can make the design process of such systems seamless while also maintaining predictability. This paper addresses the need for a C-based design framework for achieving time predictability. To this end, we propose the PRET-C language and the ARPRET architecture. PRET-C offers a small set of extensions to a subset of C to facilitate effective concurrent programming. We present a new synchronous semantics for PRET-C. It guarantees that all PRET-C programs are deterministic, reactive, and provides thread-safe communication via shared memory access. This simplifies considerably the design of safety-critical systems. We also present the architecture of a precision timed machine (PRET) called ARPRET. It offers the ability to design time predictable architectures through simple customizations of soft-core processors. We have designed ARPRET particularly for efficient and predictable execution of PRET-C. We demonstrate through extensive benchmarking that PRET-C based system design excels in comparison to existing C-based paradigms. We also qualitatively compare our approach to the Berkeley-Columbia PRET approach. We have demonstrated that the proposed approach provides an ideal framework for designing and validating safety-critical embedded systems.

Index Terms—safety-critical systems, synchronous languages, time predictability, PRET, PRET-C, WCRT, WCET.

1 Introduction

Real-time applications in automotive, aviation, and industrial automation have to guarantee not only the functionality, but also the timeliness of the results. Here, a deadline is associated with the tasks, and a failure to complete prior to this deadline may lead to catastrophic consequences. Hence, for the correctness of real-time systems, it is essential to be able to compute the worst case execution time (WCET) of the tasks in order to guarantee their deadlines.

The computation of the WCET depends on the selected programming language, the compiler tool chain, the operating system, and the target hardware. Today’s general purpose processors (GPPs) complicate the analysis by introducing speculative features that solely focus on improving the average case performance, while ignoring and sometimes worsening the ability to analyse the worst case [1]. To further complicate the analysis, real-time applications sometimes rely on a real-time operating system (RTOS) that executes over these speculative processors to manage both the concurrency and the timing constraints. The problem of concurrency managed through operating systems’s (OS) threads has been highlighted by Lee [2]: “They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism”. Understandability is lost since the programmer is burdened with ensuring correctness through complex synchronization mechanisms provided by the OS. Predictability (w.r.t. timing) is sacrificed since threads are managed by a non-deterministic scheduler. More importantly, these threads maintain full execution context during context switching. This makes OS (and RTOS) threads “heavy-weight”, in the sense that there is a significant performance penalty to be paid for control dominated multithreaded applications.

One alternative would be to design smart compilers that alleviate many complications of WCET analysis. Consider, for example, the approach taken by the WCET aware code generators such as the WCC compiler [3]. In this approach, the compiler tool chain is integrated with value analysis, loop bound analysis, cache analysis and path analysis techniques to compute the WCET estimates. Also, the tool chain is aware of the underlying memory architecture that allows the compiler to generate code such that programs can be optimised in terms of (average or worst case) execu-
tion times, energy consumption or memory footprint. Their experimental results show that the WCET of real-time applications can be reduced by 30% to 40%. However, as these compilers are intertwined with specific processors, a change of architecture leads to considerable overhead to modify the back-end code generator and the WCET analyser.

A move away from traditional RTOS based approaches and WCET aware code generators is the concept of lightweight multithreading in C—a language of choice for designing embedded systems. Here, multithreading is made feasible thanks to simple libraries and by lightweight context switching. Two prominent examples in this category are the recent SC [4] language and an earlier C-library called Protothreads [5]. Both languages are inspired by the concept of coroutines, where blocks of code (called coroutines) may have multiple entry points that transfer control to other code blocks using yield statements. Both languages rely on C macros to generate C code. SC is designed mainly for directly encoding SyncCharts [6] in C, with the goal of achieving a reduced code size in comparison to Esterel based implementations of SyncCharts. This is achieved by having a single tick function that manages the state transition between the threads by using computed goto statements. Protothreads [5] is a lightweight C library for the programming of concurrent state-machines. The main objective is to produce minimal memory footprint for embedded applications, such as sensor networks.

In this paper, we propose a solution that is significantly different from the above mentioned lightweight C libraries in the following way. Firstly, we propose a synchronous C language, called PRET-C (Precision Timed C), where lightweight threads communicate through shared memory. Our synchronous semantics ensures that shared memory access is thread-safe by construction. This addresses the issue of understandability by shifting the responsibility of shared memory management from the programmer to the semantics of the language. Our threads are lightweight and are compiled to a single function where multithreading is elicited through context switching, thanks to a tick barrier instruction called EOT (End Of Tick). By relying on a new synchronous semantics, composition is causal [7] by construction and, hence, all PRET-C programs are deterministic. Finally, predictability is an essential part of real-time computing. We also present an architecture that is tailored to execute PRET-C and maintain predictability. To guarantee that all deadlines are met, we perform static timing analysis of PRET-C. This analysis, however, is not the focus of the current paper, and has already been presented in [8], [9].

Similar to PRET-C, there exist several synchronous extensions to C such as ECL [10], and ReactiveC [11], the latter being the closest to PRET-C. However, Esterel, ECL, and ReactiveC are not designed to simplify the efforts required by a static timing analyser to guarantee predictability. E.g., Esterel does not offer native data handling. This is delegated to external function calls (for instance in C). Like other languages (ECL and ReactiveC), Esterel does not impose any restrictions on C such as recursion, dynamic memory allocations and loops bounds, which are very important for timing analysis. In contrast, PRET-C clearly states the restrictions and the language semantics (like fixed threads priorities and syntax (like user specified loop bounds) simplify the efforts required by the timing analyser.

The proposed language extensions to C are based on C macros and, hence, the standard gcc compiler can be used for code generation. Then, to achieve predictability, this code must run on a platform that provides predictable execution. The architecture we present configures a customisable RISC processor to reduce speculative features, therefore providing a predictable platform. However, this reduces the throughput. To address this issue, we present a predictable hardware accelerated platform, called ARPRET, which can execute PRET-C predictably while regaining some throughput. Our approach is inspired by the philosophy of Precision Timed Machines (PRET) [12], which provides both temporal predictability and throughput. In a detailed comparison between our approach and the PRET architecture [13] is presented in Sec. 6.

Alternatively, we can use reactive platforms such as ReMIC [14], STARPro [15] and KEP3a [16], which focus on control dominated programs and have minimal support for the execution of algorithms in C. In comparison, ARPRET is designed to execute natively C-based synchronous programs.

The key contributions of this paper are:

1) The design of a new lightweight and concurrent language called PRET-C, for the predictable programming of PRET architectures. Thanks to its synchronous semantics, PRET-C offers a very simple mechanism for achieving thread-safe shared memory communication between lightweight C-threads, not available in earlier lightweight threading libraries for C.

2) We present the semantics of PRET-C using structural operational style [17]. The main difference with earlier synchronous semantics (e.g., Esterel [18] or SL [19]) comes from the encoding of parallel threads as a fixed sequence based on the static priority of the threads.

3) We propose a hardware accelerator for PRET-C execution over soft-core processors, so that predictable execution can be achieved without sacrificing throughput. The resulting PRET architecture is called ARPRET.

The organization of this paper is as follows. In Sec. 2, we present the PRET-C language through an unmanned aerial vehicle example. Sec. 3 presents the comparison between PRET-C, Esterel, and ReactiveC.
The formal semantics is presented in Sec. 4. In Sec. 5, we present the predictable ARPRET architecture. In Sec. 6, we compare our ARPRET architecture with the Berkeley-Columbia PRET architecture. The results of the experiments are presented in Sec. 7. Conclusions are presented in Sec. 8.

2 PRET-C overview

The overall design philosophy of PRET-C and the associated architecture may be summarized using the following three simple concepts:

- Concurrency: Concurrency is logical but execution is sequential. This ensures both synchronous execution and thread-safe shared memory communication. This is the founding principle of the synchronous programming languages [7].
- Time: Time is logical and the mapping of logical time to physical time is achieved by the compiler and the WCRT analyser [9]. This will be illustrated through an example in Section 2.3.
- Design approach: The extensions to C are minimal and are implemented through C macros. Hence, PRET-C can be executed on any processor that has support for executing C programs.

2.1 PRET-C language extensions

PRET-C extends C using the nine constructs shown in Table 1. It allows the use of structures, unions and user defined macros. However, in order to guarantee the predictable execution, we impose the following four restrictions on the C language:

1) Pointers and dynamic memory allocation are disallowed to prevent unpredictability of memory allocation. This restriction is classically adopted by companies that design embedded software with C.

2) To guarantee an upper bound on the execution time, all loops must have a bounded number of iterations, which may be specified with the parameter “n” (see Table 1). If such a bound is not available, at least one EOT statement must be present in the loop body, which is guaranteed to be executed in each iteration.

3) All functions must be non-recursive (to ensure a temporal upper bound on the execution time) and must use only plain C (i.e., no PRET-C statements). This restriction will be relaxed in the future.

4) All functions and expressions must be side effects free, i.e., aside from returning/computing a value, they must not change the state of any global or static variables. This restriction is commonly imposed by coding rules found in the software industry of safety-critical systems, and is even enforced by languages like Ada [20].

The nine C extensions (presented in Table 1) are implemented as C macros, all contained in the pretc.h file that must be included at the beginning of all PRET-C programs. As a result, we only rely on the gcc macro-expander and compiler for compiling PRET-C programs. We do not modify the compiler unlike [3], where the compiler back-end is modified to generate WCET aware code.

Like any C program, a PRET-C program starts with a preamble part (#define and #include lines), followed by global declarations (reactive inputs, reactive outputs, and classical C global variables). This is followed by thread definitions, including the main thread. Both reactive inputs and outputs have a global scope and are thus visible to all threads and functions.

A PRET-C program runs periodically in a sequence of ticks. The inputs coming from the environment are sampled at the beginning of each tick. They are declared with the ReactiveInput statement. The outputs emitted to the environment are declared with the ReactiveOutput statement. Reactive inputs are read from the environment at the beginning of every tick and cannot be modified inside the program. Hence, the value of the input variables remains fixed throughout a tick. In contrast, reactive outputs may be updated by the program and can have several values within a tick. The final value of these variables (termed their steady-state value) is emitted to the environment. Reactive outputs behave exactly like normal variables in C, except that they are emitted to the environment, while normal variables are used for any intermediate computations (as in C) or for communication between threads, and are not emitted to the environment.

The PAR(T1, ..., Tn) statement spawns n threads that are executed in lock step. All spawned threads evolve based on the same view of the environment. However, unlike the usual parallel (||) operator of other synchronous languages like Esterel, where threads are scheduled in each instant based on their signal dependencies, threads in PRET-C are always scheduled based on a fixed static order. This order is determined by the order in which the threads are spawned using the PAR statement. E.g., a PAR(T1, T2) statement assigns to T1 a higher priority over T2. This fixed order allows all PRET-C programs to be causal by construction, and simplifies the complexity of the static timing analysis, as significantly fewer interleavings are needed to be explored. However, when compared to Esterel, PRET-C is less expressive w.r.t. concurrency and synchrony (See Section 3).

Parallel threads communicate through shared variables and reactive outputs. Their mutual exclusive access is achieved by ensuring that, in every instant, all threads are executed in a fixed total order by the scheduler (thanks to the fixed priorities between the threads). When more than one thread acts as a writer for the same variable, they always do so in a fixed order, ensuring that the data is consistent. Thus, the behaviour of the program remains deterministic.
However, it requires the programmer to think about the data dependencies between the threads and to insert EOTs such that shared variables are accessed in the correct order.

The EOT statement marks the end of a tick. When used within several parallel threads, it implements a synchronization barrier between those threads. Indeed, each EOT marks the end of the local tick of its thread. A global tick elapses only when all participating threads reach their respective EOTs. For illustration, consider the PRET-C statement $\text{PAR}(T_1,T_2)$. Let $T_1$ consists of $A;\text{EOT};C;\text{EOT}$, while $T_2$ consists of $B;\text{EOT};D;\text{EOT}$, where $A, B, C, D$ are blocks of C statements. The thread interleaving within a given tick is shown in Figure 1, where the EOTs mark specific context switching points. The code between the EOT statements of each thread is executed in sequence during a tick. The semantics of PAR and EOT is further elaborated through SOS style semantics in Section 4 (see Figure 5).

EOT enforces the synchronization between the parallel threads by ensuring that the next tick is started only when all threads have reached their EOT. It is also similar to the pause statement of Esterel and the yield statement of coroutines. It also enables the precise computation of the program's WCRT (defined as the maximal duration of a global tick), from the execution time of all the computations scheduled during a global tick. This WCRT analysis is presented in [8], [9].

During the first tick, the program executes line 1, and stops at the EOT. During the second tick, $x$ is assigned the value 1 and the program enters the abort body. Before executing line 5, the abort's preemption condition is evaluated. Since the expression $x$ evaluates to a non zero value ($x$ having been assigned the value 1), the body of the abort is terminated and lines 5 to 9, including the bounded loop, are never executed.

The await (exp) statement evaluates the expression exp first, and if the expression evaluates to a non zero value, then the control simply jumps to the following instruction. Otherwise, the thread finishes its local tick, and in the next tick it re-evaluates the expression exp.

The thread statement is used to spawn threads. The usage is presented in Section 2.2 through a PRET-C example.
bound on the number of iterations of each loop. Finding a tight bound is generally very hard [21], [22], and sometimes even impossible (e.g., the loop may just be infinite). Instead, we propose the following pragmatic solution. PRET-C distinguishes two kinds of loops, the instantaneous loops and the non-instantaneous loops:

**Instantaneous loops:** Their body contains only regular C statements. In order to ensure bounded execution time, the programmer must indicate an upper bound on the number of iteration of such loops. This extension of while loop is supported using the syntax "While\((exp,n)\)\{P\}". The body \(P\) will be executed \(k\) times, where \(k = \min(n, \ell)\) and \(\ell\) is the number of iterations of the equivalent pure C while loop i.e., "while\((exp)\)\{P\}" Potentially, \(\ell\) could be equal to \(+\infty\) (if the loop diverges, which can easily happen in C) and hence the \(\min\) operator guarantees a bounded execution time. It is therefore the responsibility of the programmer to give a correct value to \(n\).

Similarly, bound on the for loop can be specified using the syntax "For\((init,exp,inc,n)\)\{P\}". E.g., For\((i=0,i<x,i++,10)\)\{P\}. During the timing analysis, we compute the WCRT of the loop by multiplying "\(n\)" by the WCRT of \(P\) (plus some overhead for the loop management).

**Non instantaneous loops:** Their body must contain at least one EOT. A conservative approach is employed to determine whether all timed loops execute at least one EOT in every iteration. Firstly, a control flow graph (CFG) is extracted from the binaries of a given PRET-C program. Secondly, to simplify the CFG all function calls are inlined (for more details see [23]). Finally, using depth-first search over the graph (linear time complexity) all strongly connected components are identified and checked for instantaneous loops. During the search all data computations are ignored to reduce the analysis time. Any program with instantaneous loops is rejected, just like in Esterel.

We present multiple examples of instantaneous and non-instantaneous loops in the Appendix C.

### 2.2 An Unmanned Aerial Vehicle example

**Fig. 2. UAV example overview**

To illustrate PRET-C, we present in Figure 2 an overview of an Unmanned Aerial Vehicle (UAV) controller. This example is a pedagogic abstraction of a benchmark in PapaBench [24]. The example has four tasks. The navigation controller task calculates...
the desired angle of the flaps and the desired engines speed (speedL and speedR). This is done based on the current altitude, speed, and the GPS position of the vehicle. The altitude controller task controls the duty cycle of the flaps. This is based on the value of the variable angle, which represents the desired angle of the flaps. The engine controller task controls the pulse width modulation (PWM) duty cycle of the left and right engines based on the values of speedL and speedR outputs. The failSafe task makes sure that the value of angle always operates within a safe range.

The UAV operates in two modes. In the flight mode the UAV takes a preprogrammed route. In contrast, in the landing mode, the UAV shuts down its engines and glides down to land. There is a strict constraint that the time from capturing the inputs to determining the outputs should not exceed 0.1 seconds.

We present the PRET-C model of the UAV example in Figure 3. The program starts by including the pretc.h file (line 1). Then it includes the uav.h file containing the user defined functions (line 2). Reactive inputs/outputs are declared (lines 4 to 11), followed by C global variables (lines 13 to 15), which are used for sharing data between threads and must be initialised. Finally all the threads are defined (lines 38 to 68).

During the first tick of the main thread, it spawns four threads (line 67). The textual order assigns navCnt thread the highest priority followed by engCnt thread, the altCnt thread, and the failSafe thread. As explained in Section 2, all threads are executed in fixed order until they finish their local ticks (marked by EOT statements).

First, the navCnt thread reads the data from the reactive inputs mode, altitude, airSpeed and GPS. Next, it evaluates the preemption condition (line 26) before executing the strong abort body (lines 20 to 25). If the preemption condition evaluates to false, user defined functions (lines 21 to 23) to calculate the desired flap angles and engine speed are invoked. This data is written on to the global shared variables angle, speedL and speedR. Finally, it executes the EOT statement (line 24), reaching the local tick of this thread. However, if the preemption condition (line 26) evaluates to true, the abort body is preempted and the thread continues execution from line 27.

When the vehicle is in landing mode, it abruptly shuts down the engines (lines 27 and 28). Control enters the weak abort body (lines 30 and 33) and calculates the new angle for gliding. Finally, it executes the EOT (line 32), completing its local tick. When the weak abort preemption condition (line 34) evaluates to true, the while loop terminates its body and the control continues from line 19. However, if the preemption condition evaluates to false, the thread finishes its local tick (line 32). This allows the next thread (engCnt) with the next highest priority to start its execution.

In the engCnt thread, user defined function (line 40) is invoked to calculate the desired flap angles. Then the EOT statement (line 41) is executed, finishing its local tick. Next, the altCnt thread reads the shared variable angle and checks if the flap angles should be updated (lines 49 to 54). Finally, it executes the EOT (line 55), finishing its local tick. Next, the failSafe thread starts its execution. Line 61 ensures that the value of flaps always lies within a safe range. Then, it executes the EOT (line 62), reaching its local tick. Since all threads have finished their local ticks, this marks the global tick of the program. The PRET-C program now starts a new tick, and resumes all threads from their previous local ticks.

2.3 Verifying Timing Requirements
PRET-C programs have a formal model of computation based on the synchrony hypothesis. This hypothesis abstracts time into discrete ticks so that outputs are updated instantaneously relative to the inputs. This hypothesis requires that the idealized synchronous program executes infinitely fast relative to its environment. For all practical purposes, the synchrony hypothesis can be respected if the maximum length of computation of the reactive function is less than the minimum inter-arrival time of events from the environment. For our running example, this minimum interval-arrival of events is specified as a timing constraint where the time from from capturing the inputs to determining the outputs should not exceed 0.1 seconds. For our PRET-C program and given the MicroBlaze (MB) processor (details about the processor are presented in Section 5), our WCRT timing analyser [9] finds out that the worst case reaction time (WCRT) of the program is 1042 machine cycles. The maximum operating frequency of MB is 200 MHz [25]. Thus, for our example, running MB processor at any frequency above 95.9 MHz (0.1/1042) will satisfy the timing constraint.

3 Comparison with Esterel, Reactive C and Synchronous C
This section illustrates the language through a set of examples and compares it with Esterel [26], Reactive C [11] (denoted RC), and Synchronous C (denoted SC). We are not comparing with the ECL language [10] since it is almost identical to Esterel semantically. We start by providing a qualitative comparison of the languages as shown in Table 2.

Firstly, the nature of synchronous parallel is different. In Esterel and SC, it is possible to execute the same set of threads with different execution orders in different instants provided that this order respects the producer consumer dependencies of signals. The Esterel compiler computes an interleaving
of the threads that varies depending on the signal dependencies. The equivalent Esterel sequential code of Figure 4(a) is “emit (A); emit B (?A); pause; emit B (7); emit A (?B+1);” In contrast, the equivalent PRET-C sequential code of Figure 4(b) is “A=0; B=A; EOT; A=B+1; B=7.” This difference in the sequential code, explains signals/values shown in Figure 4(c).

In contrast, both in RC and PRET-C, all threads always execute based on a fixed static order. Moreover, Figure 4 also illustrates the fact that the same behaviour cannot be obtained by a direct translation, i.e., by replacing the pause statement with the EOT statement.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Esterel</th>
<th>RC</th>
<th>SC</th>
<th>PRET-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutativity of evaluation</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Communication between threads</td>
<td>signals</td>
<td>signals &amp; variables</td>
<td>signals &amp; variables</td>
<td>variables</td>
</tr>
<tr>
<td>Instantaneous broadcast</td>
<td>yes</td>
<td>yes/no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Signals/variables</td>
<td>single</td>
<td>multiple</td>
<td>multiple</td>
<td>multiple</td>
</tr>
<tr>
<td>Types of aborts</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Types of suspend</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Traps</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Non-causal programs</td>
<td>possible</td>
<td>possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
<tr>
<td>Dynamic Processes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Compilation</td>
<td>complex</td>
<td>complex</td>
<td>macro-expansion</td>
<td>macro-expansion</td>
</tr>
</tbody>
</table>

TABLE 2
Qualitative Comparison with Esterel and RC

Secondly, signals are the only means of communication between threads in Esterel and RC. In Esterel, a signal is either present or absent in any instant. Additionally, a signal may carry a value, and multiple emissions on the same valued signal is supported using a combine operator.

RC relaxes these restrictions by allowing multiple value emissions of the same signal, without a combine operator, where the last emitted value overrides the previous emissions. It also allows the emitted signal to be reset within an instant using the reset statement. This is not possible in Esterel. One problem with RC is that signal checking is done dynamically and once a signal has been read, if a future emission occurs, then a run-time exception is raised. This approach can lead to unpredictable run-time behaviour, which is undesirable for embedded systems.

In contrast with both Esterel and RC, all communications between PRET-C threads are through C shared variables (reactive outputs and global variables), similarly SC threads communicate through shared variables or signals (encoded again as variables). Reactive input variables are read from the environment in the beginning of an instant and their value doesn’t change during the whole tick. In contrast, a reactive output variable can have multiple values within an instant due to multiple threads writing into this variable. Only the final value is emitted to the environment (this is similar to the RC approach of allowing multiple writers for signals). Unlike both Esterel and RC, all PRET-C (and SC) threads have unrestricted access to all shared variables. Since all threads execute in a fixed order and all reads and writes are atomic, the program remains deterministic; SC allows dynamic changes to thread priorities, but the changes are based on compile-time decisions. Hence, any SC program is still deterministic. In contrast, RC allows the access to shared variables. However, the duty of maintaining data-coherency is left to the programmer.

The notion of instantaneous broadcast is different. In Esterel, once a signal is emitted, all other threads can test this value instantaneously. This is not directly possible in RC as the parallel operator is mapped to a fixed sequence. PRET-C too cannot support the notion of instantaneous broadcast. SC allows instantaneous communication, but when using this, it is up to the programmer (or a code synthesizer) to ensure that write-read dependencies are met by assigning thread priorities accordingly. Another distinction is that all preemptions in PRET-C are based on the evaluation of the condition at two possible control points. First, when the control enters the abort body for the first time. Second, either at the beginning or the end of a local tick. Also, Esterel/RC can have non-causal [7] composition of threads due to feedback cycles. These must be rejected by the compiler and the associated causality analysis results in large compiling overheads. PRET-C programs are causal by construction due to the semantics of the parallel operator. Hence, compilation, and static analysis of PRET-C programs is much simpler, and so are SC programs with only static priorities (dynamic priorities would require a more involved WCRT analysis [27]).

4 Semantics of PRET-C

In this section, we present the semantics of PRET-C using structural operational style [17]. The proposed semantics resembles the semantics of earlier synchronous languages like Esterel [18] and the SL language [19]. The main difference stems from the en-
coding of parallel as a fixed sequence based on the priority of threads. This simplifies the language and ensures the determinism of all PRET-C programs with arbitrary data dependencies. To formally describe the semantics of PRET-C, we define a small set of kernel statements, as in [28]. PRET-C is first compiled into this kernel and the behaviour of the language is described by formally describing the behaviour of the kernel statements.

4.1 The Kernel language

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>Terminates instantaneously without doing anything.</td>
</tr>
<tr>
<td>EOT</td>
<td>Marks the completion of a reaction (global tick) when all local ticks have been reached.</td>
</tr>
<tr>
<td>t;u</td>
<td>Execute t followed by u.</td>
</tr>
<tr>
<td>if(c){t}else{u}</td>
<td>Conditional statement</td>
</tr>
<tr>
<td>while(c) {t}</td>
<td>t must have at least one EOT, or can guarantee the loop bound statically.</td>
</tr>
<tr>
<td>PAR(t,u)</td>
<td>Logical parallel execution of t and u with higher priority for t.</td>
</tr>
<tr>
<td>end</td>
<td>When used inside a PAR statement, it terminates both threads.</td>
</tr>
<tr>
<td>v=f(...)</td>
<td>Computes f and then assigns the result to variable v. If cannot contain any EOT or PAR statements, and cannot be recursive.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language extension</th>
<th>Mapping to the kernel language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactiveInput I</td>
<td>mapped to a global variable I</td>
</tr>
<tr>
<td>ReactiveOutput O</td>
<td>mapped to a global variable O</td>
</tr>
<tr>
<td>PAR(T1,...,Tn)</td>
<td>PAR({T1},...,{PAR(Tn-1,Tn)}) nested PAR statements.</td>
</tr>
<tr>
<td>EOT</td>
<td>EOT</td>
</tr>
<tr>
<td>abort(exp){p}</td>
<td>PAR({while(!exp){EOT}; end;}, {p; end;})</td>
</tr>
<tr>
<td>weakabort(exp){p}</td>
<td>PAR({p; end;}, {while(exp){EOT}; end;})</td>
</tr>
<tr>
<td>await(exp)</td>
<td>while(!exp){EOT;}</td>
</tr>
<tr>
<td>thread T(){p}</td>
<td>mapped to a label T:{p}</td>
</tr>
<tr>
<td>While(exp,n){p}</td>
<td>while(exp &amp;&amp; (!cnt≤n)){cnt++; p;}</td>
</tr>
<tr>
<td>For(init,exp,inc,n){p}</td>
<td>init; while(exp,n){p;inc}</td>
</tr>
</tbody>
</table>

| TABLE 3 | Kernel statements of PRET-C |

We introduce a kernel PRET-C language, where statements are summarized in Table 3. The kernel language does not consider the entire C language, and mainly focuses on the PRET-C extensions.

4.2 Structural Translations

<table>
<thead>
<tr>
<th>Language extension</th>
<th>Mapping to the kernel language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactiveInput I</td>
<td>mapped to a global variable I</td>
</tr>
<tr>
<td>ReactiveOutput O</td>
<td>mapped to a global variable O</td>
</tr>
<tr>
<td>PAR(T1,...,Tn)</td>
<td>PAR({T1},...,{PAR(Tn-1,Tn)}) nested PAR statements.</td>
</tr>
<tr>
<td>EOT</td>
<td>EOT</td>
</tr>
<tr>
<td>abort(exp){p}</td>
<td>PAR({while(!exp){EOT}; end;}, {p; end;})</td>
</tr>
<tr>
<td>weakabort(exp){p}</td>
<td>PAR({p; end;}, {while(exp){EOT}; end;})</td>
</tr>
<tr>
<td>await(exp)</td>
<td>while(!exp){EOT;}</td>
</tr>
<tr>
<td>thread T(){p}</td>
<td>mapped to a label T:{p}</td>
</tr>
<tr>
<td>While(exp,n){p}</td>
<td>while(exp &amp;&amp; (!cnt≤n)){cnt++; p;}</td>
</tr>
<tr>
<td>For(init,exp,inc,n){p}</td>
<td>init; while(exp,n){p;inc}</td>
</tr>
</tbody>
</table>

| TABLE 4 | Structural translations of PRET-C extensions into the kernel statements |

Table 4 presents the structural translations for mapping the language extensions (Table 1) into the kernel statements (Table 3). ReactiveInput and ReactiveOutput are mapped to global variables. PAR with n threads is translated into n-1 nested PAR kernel statements (row 3 of Table 4).

Traditionally, the semantics of an abort is captured using traps [26]. The abort body is preempted when an exception is raised by using an exit statement. This exception can be raised at any time, during the execution of the body. However, in PRET-C we have simplified the behaviour of an abort. The abort body can only be preempted at the tick boundaries. This restriction allows us to translate the aborts into a PAR statement, and an exception is raised using the end statement. This mapping also reduces the need for an extra kernel statement.

PRET-C supports both strong and weak aborts. In both cases, an abort is translated into two threads. In the strong abort case, the first thread checks the preemption condition while the second thread captures the abort body. If either of the threads terminate, an exception is raised using end statements (row 5 of Table 4). In the weak abort case, the threads are reversed. The first thread captures the abort body, and the second thread checks the preemption condition (row 6 of Table 4). Thanks to the higher priority of the first thread over the second one, the semantics of the strong and weak abort is enforced. This is further explained in operational semantics of PAR (Section 4.3).

await(exp) is mapped into a while loop with EOT as the body (row 7 of Table 4). It evaluates the expression every tick until the expression evaluates to a non zero value.

thread T(){p} is mapped into a program point T: with body P (row 8 of Table 4).

All bounded for loops are translated into bounded while loops (row 10 of Table 4). Then, the bounds are translated into C code according to the simple translation (row 9 of Table 4). In this translation, n is the loop bound and cnt tracks the number of iterations completed by the loop. Since we want to force the bound on each loop, each counter cnt must be unique.

4.3 Rewrite rules

Program transitions are represented by SOS rules of the form $E : t \xrightarrow{k} E'$, $t'$ where:

- $t$ is a term that consists of any arbitrary composition of kernel statements.
- $t'$ is the residual of a term after the transition has been taken.
- $I$ is the set of reactive inputs.
- $E$ is the status or valuations of a set of variables (both global and local).
- $E'$ is the status of the same set of variables after the transition has been taken.
- For efficient encoding of transitions, we use completion codes ($k$). It is produced during a transition.
from \( t \) to \( t' \), as in [18].

- If \( k = 0 \), then this transition represents that a \textit{nop} has been executed.
- If \( k = 1 \), then this transition represents that an EOT has been executed.
- If \( k = 2 \), then this transition represents that an \textit{end} has been executed. This code captures preemptions by the abort statements.
- If \( k = 3 \), then this transition represents that a C instruction (assignment, if statement, or a loop) has been executed.

Figure 5 presents all the rewrite rules for the kernel statements. We now describe each rule in detail:

\textbf{The nop statement}: terminates instantaneously. It does not have a residual term, i.e., the program has terminated. It is captured by rule (NOP).

\textbf{The end statement}: is rewritten into nop and is captured by rule (END). Compared to the nop statement, end has a different completion code. This difference in the completion codes is used to differentiate between a termination or a preemption of threads.

\textbf{The EOT statement}: completes the local tick of the program; and this is captured by a completion code of 1. This is rule (EOT).

\textbf{The sequence operator}: The right branch \( u \) is executed sequentially after the left branch \( t \). When the left branch terminates with a \textit{nop} \( (k = 0) \), the sequence simply rewrites into the right branch (SEQ2). However, if the left branch does not terminate with \( k = 0 \), then the completion code of sequence operator is same as the completion code of the left branch (SEQ1).

\textbf{The conditional statement}: There are two rules corresponding to the cases when the guard condition evaluates to true (THEN) or false (ELSE).

\textbf{The assignment statement}: A variable can be assigned a value returned by a function or the result of the evaluation of an expression (which can also be encapsulated as a function). The (ASSIGN) rule states that, when such a statement is executed, the value of the variable \( v \) is updated by the return value of the function in the environment \( E \).

\textbf{Loops}: Loops are captured by two simple rewrite rules. When the guard condition evaluates to true, the loop body will be unrolled once (WHILE1). When the guard condition evaluates to false, the statement simply terminates (WHILE2). In both cases, the completion code is 3.

\textbf{The PAR statement}: The parallel execution of the terms \( t \) and \( u \), where the priority of \( t \) is higher than that of \( u \), is captured by first executing all statements of \( t \) (by rule PAR1) until the local tick of \( t \) is reached (by rule PAR2). Rule PAR2 pauses \( t \) until \( u \) reaches its EOT (that is, as long as the completion code of \( u \) is 3). For PAR1, the second premise \( (E : u \xrightarrow{k} E' : u') \) is needed to simplify the proof for determinism (presented in Appendix B).

The \textit{PAR} pauses with a completion code of 1 when both threads have reached their EOT (rule PAR3). If \( t \) is at an EOT while \( u \) terminates, then \textit{PAR}(t, u) also terminates with a completion code of 1 (rule PAR4). However, if \( t \) terminates, then \textit{PAR}(t, u) just becomes \( u \) and terminates with \( 3 \) (rule PAR5).

If either \( t \) or \( u \) is at an end statement, then \textit{PAR}(t, u) terminates with a completion code of 3 (rule PAR6). This rule is used to capture the preemption behaviour of aborts. Indeed, aborts are translated into kernel statement \textit{PAR} (rows 5 and 6 of table 4).

We present an illustration of the semantics in the Appendix A.

4.4 Reactivity and Determinism

\textit{Definition 1}: The \textit{reaction} of a program \( t \) in an instant is denoted as \( E : t \xrightarrow{k} E' : t' \) if there exists a sequence of transitions such that \( E : t \xrightarrow{l} E_1 : t_1 \cdots E_n : t_n \xrightarrow{k} E' : t' \) with \( k \in \{0, 1\} \) and \( \forall 1 \leq i \leq n, i \neq k \notin \{0, 1\} \).

\textit{Definition 2}: A program \( t \) is \textit{reactive} if, for any data set \( E \), there exists at least one \textit{reaction} given the set of inputs \( I \) i.e., the program doesn’t deadlock in any state.

\textit{Definition 3}: A program \( t \) is \textit{deterministic} if, for any data set \( E \), and any inputs \( I \) there exists at most one \( (E', k', t') \) such that \( E : t \xrightarrow{k'} E' : t' \).

\textit{Theorem 1}: All PRET-C programs are reactive, i.e., \( \forall t, \forall E, \forall I \), there exists \( t' \) and \( E' \) such that \( E : t \xrightarrow{k} E' : t' \) and \( k \in \{0, 1\} \).

\textit{Theorem 2}: All PRET-C programs are deterministic, i.e., \( \forall t, \forall E, \forall I \) such that there exists at most one \( (E', k', t') \) such that \( E : t \xrightarrow{k'} E' : t' \).

We present the proofs for reactivity and determinism in Appendix B.

5 Executing PRET-C

PRET-C is a lightweight multi-threaded language. To guarantee the timing predictability, the underlying hardware must be amenable to timing analysis. General-purpose processors are highly speculative and are focused on improving average case performances. Such architectures with speculative features make static timing analysis very challenging [1]. To solve this problem, one approach would be to take a simple customisable RISC processor and turn off speculative features (referred to as the basic platform). This provides a predictable platform, and simplifies the timing analyser. However, turning off the speculation decreases the throughput of a program. We address this issue by presenting a hardware accelerator which is tailored to accelerate the PRET-C scheduling overheads, and thus improving the throughput. This hardware accelerator can be attached to a predictable
RISC processor, creating a new predictable platform (referred to as ARPRET platform). The rest of this section presents more details about the basic platform and the ARPRET platform.

5.1 The basic platform
To execute PRET-C, we have selected a customisable processor called MicroBlaze (MB) [25]. To maintain predictability we have reduced speculative features such as out-of-order execution and caches. We have then implemented the thread scheduling very efficiently in software, thanks to a CEC-like [29] linked-list based scheduler. This scheduler is responsible for thread creation, context switching between threads, and termination of threads according to the PRET-C semantics. In the following section, we present a hardware accelerator for reducing this scheduling overhead.

5.2 ARPRET platform
Figure 6 shows the setup of the ARPRET platform. It consists of a MicroBlaze soft-core processor (MB) [25] coupled with a hardware accelerator called the Predictable Functional Unit (PFU). Like in Section 5.1, we first customise MB to reduce speculation. However, we regain some throughput thanks to the PFU. The PFU is connected to the processor using two fast simplex links (FSL) [25]. FSL allows for predictable and fast communication between the PFU and the processor. This is essential to maintain a predictable platform with low communication overheads.
orchestrates the creation, termination and suspension of all threads. During the execution of PAR, MB sends multiple thread contexts to the PFU. Each context includes the thread’s Program Counter (PC), Priority (TP), status (dead or alive, called TDA), suspension status (TSP), and local tick status (TLT), all stored in the thread table. Based on these thread statuses, the scheduler sends the next thread’s PC, when requested by the MB processor. Similarly, during the execution of an abort statement, all preemption contexts are stored in the abort table. The scheduler then evaluates all preemption conditions at the tick boundaries, and based on the nesting of aborts, the next PC that needs to be executed is sent to MB.

When a thread executes an EOT, MB sends a request to the PFU’s control logic to assert the local tick status bit for this thread. Then, the MB requests the scheduler for the next thread’s PC to be executed. During every request, MB is blocked, and only resumes when the new PC is received. For a local tick, this always takes exactly seven clock cycles, making it perfectly predictable. Concerning the global ticks, PFU can be configured for two different execution strategies, either with a variable tick length for better performance, or with a constant tick lengths for full predictability. During the constant execution mode, MB awaits for the worst case tick length to expire before starting the next global tick. For any given PRET-C program, this constant tick length is computed by a static WCRT analyser [9].

5.3 Comparison between the basic and the ARPRET platforms

A key difference between the basic and the ARPRET platform is the implementation of the scheduling. In the basic platform, the scheduler is implemented in software using a linked list based approach [29]. In contrast, in the ARPRET platform, the scheduler is implemented in hardware using the PFU (presented in Figure 6). Using Table 5, we illustrate the difference in execution times as the program context switches at EOT, where the current thread’s program counter (PC) is stored and the control is resumed from the next thread’s PC.

For executing the EOT statement, the basic platform takes 25 clock cycles, while the ARPRET platform takes only 11 clock cycles. This difference occurs mainly during step 3. During this step, in the basic platform, the next thread’s context is updated using a C—based linked list operation (currentThread = currentThread->next, which requires five assembly instructions as follows:

```
addik r3, r0, 792 // 1 clks
swi r3, r4, 0 // 2 clks
lwi r3, r0, 6884 // 2 clks
lwi r3, r3, 12 // 2 clks
swi r3, r0, 6884 // 2 clks
```

In contrast, in the ARPRET platform, as steps 1 and 2 are in progress, the PFU in parallel computes the next thread’s PC. Thus, this step does not cost ARPRET any clock cycles. This efficiency of ARPRET is also observed during other operations such as thread creation, termination and during the evaluation of the preemption conditions. Later, in the results section, the differences between the two platforms are compared during the WCRT computation of a PRET-C program.

6 Comparison with the Berkeley-Columbia PRET approach

We present a qualitative comparison between our approach and Berkeley-Columbia’s approach [13] are to provide predictability and simplify WCET analysis, while maintaining throughput. Berkeley-Columbia’s PRET provides support at the ISA-level, while our work proposes a new programming model that supports PRET. Their work currently excels on the architectural front, while we have been exploring the language side. Hence, it is not easy to compare the two approaches quantitatively. We present a qualitative comparison between our approach and the Berkeley-Columbia approach, which is summarized in Table 6.

The Berkeley-Columbia group has recently developed a PRET processor called PTARM [13]. It has a thread interleaved five staged pipeline that is based on the ARM ISA. It also contains an advanced DRAM controller that is predictable and significantly reduces the worst case memory access latency [30]. Most importantly, they have introduced the notion of time at the ISA level with the deadi instruction to achieve precise timing of each segment of the code. Also,
deadlines are specified at key points to ensure mutual exclusive access to shared memory.

In comparison, ARPRET executes threads sequentially between logical instants. Having a notion of tick abstracts away physical time while also simplifying timing analysis [9]. Also, calculating the length of the tick is far simpler than calculating the deadline values for deadi instructions, and is performed automatically by a compiler instead of manually by the programmer. Our hardware is a co-processor that can be used with any customizable processor. This, in our opinion, is a more flexible approach for the design of PRET machines than a tailored architecture, like PTARM [13]. In contrast, the Berkeley-Columbia researchers have made progress in several directions such as the development of a predictable DRAM controller [13], [30]. More recently, they proposed new ISA extensions which are more expressive than the deadi instruction [31]. However, the problem of verifying the timer values is exactly the same as verifying the deadline values.

One important difference between the two approaches is how they satisfy the given timing deadlines. In the Berkeley-Columbia approach, the values of the deadi instructions are directly derived from deadlines and subsequently are verified using a static timing analyser. In contrast, our approach relies on the concept of logical time through EOTs and then checks for the violation of the deadlines using a static timing analyser. If the program fails to meet the deadline, then, in both approaches, the programmer must revisit the program to identify the most computing intensive parts, which must then either be optimized or be broken by inserting additional EOT/deadi statements.

7 Benchmarks and results

In this section, we present a set of experiments to evaluate the performance of the PRET-C language and the ARPRET architecture. In the first set of experiments, we assess the effect of the proposed hardware accelerator (PFU). We compare the execution throughput of PRET-C on the ARPRET platform (section 5.2) with that of the basic platform (section 5.1).

In the second set of experiments, we compare the execution of PRET-C (on the basic platform) with that of Protothreads, SC, and Esterel. Here, we compare the execution times and memory footprint over a set of benchmark programs with high degree of concurrency and preemption.

For these experiments, we have selected a set of benchmarks from Estbench [32], which have been mapped into PRET-C. We have also designed two new examples: a producer consumer example from [33] and an autonomous robot performing obstacle avoidance. The latter uses a single sonar sensor to calculate the distance, and uses two motors to control the direction and speed of the robot. All examples and their characteristics are presented in Table 7. The first column states the name of the example followed by its description (column 2). The number of threads and the number of lines in PRET-C code is presented in columns 3 and 4 respectively.

### Table 7

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Threads</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRO</td>
<td>simple synchronous program</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>synchronises communication between threads</td>
<td>4</td>
<td>51</td>
</tr>
<tr>
<td>Reactor Control</td>
<td>control of rods in a reactor</td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>Producer-Consumer</td>
<td>bounded buffer problem</td>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>Smokers</td>
<td>cigarette smokers problem from computer science</td>
<td>4</td>
<td>111</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>obstacle avoidance</td>
<td>4</td>
<td>280</td>
</tr>
</tbody>
</table>

7.1 Benchmarking

The benchmarking process was carried out as follows. Figure 7 shows the WCRT of two different but predictable execution platforms for PRET-C. Firstly, we generated code for the basic platform (MB only, with no PFU). This data is depicted by the Basic bars in Figure 7. Then, for the same set of benchmarks we generated code for the ARPRET platform (MB with PFU). This data is depicted by the ARPRET bars. Then, using the static timing analyser, we have calculated the WCRT of each benchmark for both platforms. On average, compared to the basic platform, the ARPRET platform decreases the WCRT by 26%. The biggest difference is seen in the Robot Sonar example. It has lot of context switching between the threads, and the use of PFU clearly gives ARPRET an advantage over the Basic. In contrast, the Smokers example has a lot less context switching. Hence, the WCRT values for both platforms are similar.

We now compare the execution of PRET-C with Protothreads, SC, and Esterel on the basic platform. For each benchmark, we preserve behavioural equivalence when translating PRET-C programs into Protothreads, SC, and Esterel.

- In the case of Protothreads, we have synchronized threads using the yield construct. This is similar to PRET-C’s EOT instruction. We also
force global tick synchronization to ensure a synchronous execution like in PRET-C.

- In the case of SC and Esterel, it was fairly easy to translate PRET-C programs into SC and Esterel. Since they were synchronous languages too.

Since Protothreads and SC are implemented using C, we converted Esterel programs into C using the CEC compiler. Then using MB as the target processor, we measured the reaction time achieved by these languages. In Figure 8, we present the WCRT for each benchmark. On average, PRET-C consumes about 20%, 50%, and 74% less computation time when compared to Protothreads, Esteral, and SC respectively. Interestingly, for the Reactor Control example, ProtoThreads gives slightly (less than 2%) better WCRT than PRET-C. This may be due to the gcc compiler optimisation. Otherwise, PRET-C is more efficient than Protothreads, as seen in the rest of the examples. Overall, SC achieves the least throughput. This is due to its implementation of the thread scheduling by calling a function, where the overhead of the function call is proportional to the number of threads. In contrast, the context-switch in PRET-C is implemented using the linked-list based scheduler which has a constant context switching overhead.

In summary, these results reveal that PRET-C yields significantly more efficient code compared to Esterel. This gain is due to the compilation overheads during the Esterel to C translation. However, compared to SC, PRET-C generates slightly larger code (by only 4%) that is mostly equivalent to Protothreads.

In summary, these results reveal that PRET-C yields significantly more efficient code compared to all others in both the average and worst case. Also, the memory usage of PRET-C is significantly smaller than Esterel and is similar to SC and Protothreads.

8 Conclusions

We have presented the new language PRET-C, targeting real-time embedded systems, by simple synchronous extensions to C. PRET-C provides constructs for logical time, synchronous concurrency, and pre-emption. It also offers deterministic access to shared memory, such that all PRET-C programs are causal [7] and thread-safe [2] by construction. We have also presented a synchronous semantics for PRET-C, which is different from existing synchronous languages, and we have conducted a detailed comparison with Esterel, SyncCharts in C (SC) and Protothreads.

We have designed a hardware accelerator to improve the worst case behaviour of PRET-C programs so that overall real-time implementation is achieved without sacrificing throughput [12]. We have benchmarked the proposed approach by comparing an efficient software implementation of PRET-C with the hardware approach. We have also compared the software approach with two other lightweight C libraries. In all cases, the proposed approach excels both in terms of worst case execution time and code size. When compared with Esterel, PRET-C achieves consistently better results.

In summary, PRET-C semantics along with our ARPRET architecture and the static timing analyser provide a predictable platform for developing safety-critical applications. As part of our future work, we are currently extending the ARPRET platform with scratchpad memories. This will allow us to qualitatively compare with PTARM.

References


Appendix A  
Illustration of PRET-C semantics

Let us consider the PRET-C program shown in Figure 10, and let us examine how the semantical rules are applied. At each step, we highlight the changes in bold font.

0 PAR(
1 \{x=1; EOT; \} ; //thread1
2 \{y=1; \} ; //thread2
3 )
4 ;

Fig. 10. PRET-C Example

Step 1: Applying rules (PAR1) and (ASSIGN) we get:

\[ E : x = 1; EOT; 3 \rightarrow_{1} E' : nop; EOT; \]

Thus, any sequential composition of reactive programs is reactive.

Appendix B

Proofs for Reactivity and Determinism

Proof: The proof for reactivity is shown by structural induction on \( t \). Rule (NOP) and rule (EOT) imply that the following reactions are reactive:

\[ E : \text{nop} \rightarrow^{0}_{1} E : \]

\[ E : \text{EOT} \rightarrow_{1}^{1} E : \text{nop} \]

While \( END \) is always rewritten into NOP by rule (END); only sequence and \( \text{PAR} \) statements are of interest, since all other statements complete with a completion code of \( k = 3 \).

1) Consider a sequence \( t = q : r \). The inductive hypothesis is that there always exist \( q' \) and \( r' \) such that:

\[ E : \text{nop} \rightarrow_{1}^{k_{q}} E : q' \quad \text{and} \quad k_{q} \in \{0, 1\} \quad (1) \]

and,

\[ E : r \rightarrow_{1}^{k_{r}} E : r' \quad \text{and} \quad k_{r} \in \{0, 1\} \quad (2) \]

Thus,

- if \( k_{q} = 1 \), we get by rule (SEQ1)
  \[ E : r \rightarrow_{1}^{k_{r}} E' : q' ; r' \]

- if \( k_{q} = 0 \), we get by rule (SEQ2)
  \[ E : q ; r \rightarrow_{1}^{k_{q} k_{r}} E' : r' \]

Thus, any sequential composition of reactive programs is reactive.

2) Next, consider the case of \( t = \text{PAR}(q, r) \), with the inductive hypothesis of Equations (1) and (2):

- if \( k_{q} = 1 \) and \( k_{r} = 1 \), we get by rule (PAR3)
  \[ E : \text{PAR}(q, r) \rightarrow_{1}^{1} E : \text{PAR}(q', r') \]

- if \( k_{q} = 1 \) and \( k_{r} = 0 \), we get by rule (PAR4)
  \[ E : \text{PAR}(q, r) \rightarrow_{1}^{1} E : q' \]

- if \( k_{q} = 0 \) and \( k_{r} = 0 \), we get by rule (PAR5)
  \[ E : \text{PAR}(q, r) \rightarrow_{1}^{1} E : \text{nop} \]

Thus, any parallel composition of reactive programs is reactive.

<table>
<thead>
<tr>
<th>Step</th>
<th>( E_{\text{before}} )</th>
<th>( E_{\text{after}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x=undefined, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
<tr>
<td>2</td>
<td>x=1, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
<tr>
<td>3</td>
<td>x=1, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
<tr>
<td>4</td>
<td>x=1, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
<tr>
<td>5</td>
<td>x=1, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
<tr>
<td>6</td>
<td>x=1, y=undefined</td>
<td>x=1, y=undefined</td>
</tr>
</tbody>
</table>

TABLE 8  
Changes to the state of the environment \( E \).
Proof: The proof for determinism is shown by structural induction on $i$. Rule (NOP), rule (END), and rule (EOT) imply that the following transitions are deterministic:

\[
E : \text{nop} \xrightarrow{i} E :
\]
\[
E : \text{EOT} \xrightarrow{i} E : \text{nop}
\]
\[
E : \text{end} \xrightarrow{i} E :
\]

Since all conditional rewrite rules are mutually exclusive, only sequence and PAR are of interest to us.

1) Consider the sequence $t = q:r$. Then $q$ can be either a NOP or not a NOP.
   - If it’s not a NOP, then only rule (SEQ1) can be applied.
   - Otherwise, only rule (SEQ2) can be applied.

Since only one rule can be applied for any given program $t$, any sequential composition of PRET-C programs will always be deterministic.

2) Next, consider the case of $t = \text{PAR}(q,r)$. If $q \xrightarrow{k_1} q’$ and $r \xrightarrow{k_2} r’$, then for every possible combination of $k_1$ and $k_2$, there is only one rule that can be applied. This is shown in the table below for all the possible cases:

<table>
<thead>
<tr>
<th>$k_1$</th>
<th>$k_2$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(PAR5)</td>
<td>(PAR5)</td>
<td>(PAR5)</td>
<td>(PAR5)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(PAR4)</td>
<td>(PAR3)</td>
<td>(PAR6)</td>
<td>(PAR2)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(PAR6)</td>
<td>(PAR6)</td>
<td>(PAR6)</td>
<td>(PAR6)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(PAR1)</td>
<td>(PAR1)</td>
<td>(PAR6)</td>
<td>(PAR1)</td>
<td></td>
</tr>
</tbody>
</table>

Since only one rule can be applied for any given program $t$, any parallel composition of a PRET-C program will be always deterministic.

Appendix C
Loops in PRET-C

PRET-C allows two types of loops, (1) instantaneous loops (with bound on the number of iterations) and (2) non-instantaneous loops (at least one EOT is executed in each iteration). See Section 2 for more details. In this section, using Table 9, we illustrate the differences in the behaviour of these loops.

Example 1 presents an instantaneous loop. The counter $i$ is initialised to 0 and is incremented by 1 at the end of each iteration. Due to the terminating expression $i<5$, the loop should have terminated after five iterations ($i=0, 1, 2, 3, 4$), but, due to the bound value of 3, the loop terminates after three iterations ($\min(5,3)$). Thus, the print statement is executed only three times resulting in the output as shown in the last column. In contrast, Example 2 does not have the bound value, but executes an EOT at the end of each iteration. Subsequently, the print statement is executed exactly once in each tick, for five ticks.

Example 3 has a bound value of 3 and an EOT is executed in each iteration. In this case, the loop terminates after three iterations (three ticks). Thus, the print statement is executed exactly once in each tick, for three ticks. Example 4 is an invalid program, because it is neither an instantaneous loop (requires a bound) nor a non-instantaneous loop (requires an EOT).

Example 5 presents an instantaneous loop. Due to the terminating expression $i<5$, the loop should have terminated after five iterations, but, due to the bound value of 3, the loop terminates after three iterations ($\min(5,3)$). Subsequently, the print statement is executed only three times resulting in the output as shown in the last column. Finally, Example 6 presents a non-instantaneous loop that executes the print statement and the EOT instruction in each iteration, for five iterations (five ticks).

| E.g. Type | PRET-C program | Output ('|' denotes tick boundary) |
|-----------|----------------|---------------------------------|
| 1         | i=0; While(i<5,3){ printf("%d",i); i++; } | 0 1 2 3 |
| 2         | i=0; while(i<5){ printf("%d",i); i++; EOT; } | 0 1 2 3 4 |
| 3         | i=0; While(i<5,3){ printf("%d",i); i++; EOT; } | 0 1 2 3 4 |
| 4         | i=0; while(i<5){ printf("%d",i); i++; } | Invalid PRET-C program |
| 5         | i=0; for(i=0;i<5;i++){ printf("%d",i); i++; } | 0 1 2 3 4 |
| 6         | i=0; for(i=0;i<5;i++){ printf("%d",i); i++; EOT; } | 0 1 2 3 4 5 |

| TABLE 9 | Loops in PRET-C |