Well formed Control-flow for Critical Sections in RTFM-core

Per Lindgren, Marcus Lindner and Andreas Lindner

Luleå University of Technology

Email: {per.lindgren,emil.fresk, marcus.lindner,andreas.lindner, emil.fresk}@ltu.se

David Pereira and Luís Miguel Pinho

CISTER / INESC TEC, ISEP

Email: {dmrpe, lmp}@isep.ipp.pt

Abstract—The mainstream of embedded software development as of today is dominated by C programming. To aid the development, hardware abstractions, libraries, kernels and light-weight operating systems are commonplace. Such kernels and operating systems typically impose a thread based abstraction to concurrency. However, in general thread based programming is hard, plagued by hazards of race conditions and dead-locks.

For this paper we take an alternative outset in terms of a language abstraction, RTFM-core, where the system is modelled directly in terms of tasks and resources. In compliance to the Stack Resource Policy (SRP) model, the language enforces (well formed) LIFO nesting of claimed resources, thus SRP based analysis and scheduling can be readily applied. For the execution onto bare-metal single core architectures, the `rtfm-core` compiler performs SRP analysis on the model, and render an executable that is deadlock free and (through RTFM-kernel primitives) exploits the underlying interrupt hardware for efficient scheduling. The RTFM-core language embeds C-code and links to C-object files and libraries, and is thus applicable to the mainstream of embedded development.

However, while the language enforces well formed resource management, control flow in the embedded C-code may violate the LIFO nesting requirement, thus correctness is left with the programmer to ensure well formed nesting (through restricted control flow).

In this paper we address this issue by lifting a subset of C into the RTFM-core language allowing arbitrary control flow at the model level. In this way well formed LIFO nesting can be enforced, and models ensured to be correct by construction. We demonstrate the feasibility through a prototype implementation in the `rtfm-core` compiler. Additionally, we develop a set of running examples, and show in detail how control flow is handled at compile time and during run-time execution.

I. INTRODUCTION

C-based programming still remains the dominant means to the development of embedded software as of today. Target platforms vary from light-weight 8, 16 and 32 bit architectures with limited CPU and memory resources, to high performance DSP, and application processors with multi-core architectures and comparably un-limited memory. However, in common, their operation can typically be defined in terms of interaction with their environment, or more specifically, in terms of reactions to internal and external events. Such reactions may operate on, and update the system state, cause/raise other internal events, and/or emit events to the environment. Requirements and properties are naturally associated to reactions (and/or reaction chains), e.g., response time, memory and energy consumption.

To aid the development of such reactive systems, hardware abstractions, libraries, kernels and light-weight operating systems are commonplace. Such kernels and operating systems typically impose a thread based abstraction to concurrency. Under this abstraction the reactive behaviour of the system is typically encoded in terms threads actively waiting for (some) event(s) to occur in order to progress execution (performing the associated reaction).

In the mainstream of C-programming libraries and operating system APIs, the programmer has to take the responsibility of manually identifying and protecting the shared resources (in order to avoid races). Moreover, the programmer has to either ensure that the emerging locking pattern never may lead to a dead-lock situation, or carefully use the underlying thread abstractions to detect dead-locks and handle them in a graceful manner. This is indeed a daunting task in its own right, as even seemingly trivial problems expressed under a thread abstraction may be plagued by errors [1]. Further adding to the general problems of threads, correctness in the context of embedded software typically involves that the non-functional requirements and properties of the system description is universally upheld. Lee goes to the length of claiming:

If we expect concurrent programming to be mainstream, and if we demand reliability and predictability from programs, then we must discard threads as a programming model.[1]

For this paper we take an alternative outset, in terms of a language abstraction, RTFM-lang (Section II-A), and in particular the RTFM-core language which provides a concurrency model for embedded C-code.

In our model, the system is expressed in terms of the familiar notions of tasks and resources (Section II-B), which provide a formal underpinning and semantics allowing e.g., StackResource Policy (SRP) based analysis and scheduling. Tasks are concurrent in our model, with Last-In-Fist-Out (LIFO) nested critical sections, where a critical section is defined by claiming a single unit-resource. For the execution onto bare-metal single core architectures, the `rtfm-core` compiler performs the necessary SRP analysis and generates plain C-code with inlined scheduling primitives. The rendered executable is deadlock free, and exploits the underlying interrupt hardware for efficient scheduling. Designed to be

applicable to the mainstream of embedded programming, the RTFM-core language embeds C-code and allows linking to C-object files and libraries (Section II-C). The `rtfm-core` compiler treats the embedded C-code literally (without any structural or semantically analyses thereof) and is thus relying on error detection and reporting by the backend C compiler for the embedded code.

While well formed (LIFO) nesting is enforced by the RTFM-core language (and thus correct by construction), control flow in the embedded C-code may violate the well formedness and thus lead to incorrect models. Such errors can neither be detected by the `rtfm-core` compiler (acting as a pre-processor), nor by the backend C-compiler (as the C code may well be correct w.r.t. to the C language, while still violating the well formedness of resource nesting), hence correctness w.r.t. well formed nesting has until now been put at the hands of the programmer. In effect, this implies restricting the use of control flow constructs in the embedded C code, upholding single entry and exit points for the critical sections. This is non-satisfactory for two reasons, firstly, the a program passing compilation may be incorrect w.r.t. to the execution model, and secondly, the expressiveness of the programming model is impeded.

In this paper we propose a solution these problems, by lifting a subset of C-code constructs into the RTFM-core language. Our contributions ensures both well formed nesting, while at the same time maintaining the full flexibility of the C-based control flow. In Section III we review the C-based control flow mechanisms. In Section IV, we identify and approach the problems of C-based control flow. We propose a set of extensions to the -core language, and define its relation to resource management under the SRP model of the RTFM-core language. Furthermore, we present their implementation in the `rtfm-core` compiler and demonstrate their feasibility and rendered C code output on a set of representative examples.

In Section VI, we discuss related and future work, and in Section VII the contributions of the paper is concluded.

II. BACKGROUND

A. Real-Time For the Masses

Real-Time For the Masses (RTFM) is a set of experimental languages and tools, designed to the aim of facilitating software development in general and real-time embedded software in particular. The RTFM-core language (referred to as -core interchangeably) is as the name suggest, minimalistic and captures only concurrency and timing constructs. The -core language can be used either to it's own right (as a programming model) or as an intermediate format for other front-ends providing further abstractions, and/or capturing domain specific aspects. An example thereof is the object oriented fronted, RTFM-cOOre. In [2], the language was first outlined (in the context of supported systems with mixed criticality) and later implemented and used as an outset for the course in Compiler Construction [3]. The `rtfm-core` compiler renders C code and inlines references to scheduling primitives of the target run-time system. As of today, run-time

systems are provided for bare metal execution (the RTFM-kernel, currently available for the ARM-Cortex range of MCUs), and hosted environments (PTCORE for Pthread based systems, with adaptations for utilising OSX/Linux specifics, and WINCORE using the Win-32 thread API, applicable to e.g., by Windows 7 and Windows 8).

B. Stack Resource Policy & The RTFM-Kernel

The Stack Resource Policy (referred to as SRP interchangeably) [4] is a scheduling protocol for single-core processors. In short, a system is defined by a set of tasks (jobs) $J = j_1 \dots j_n$, with $p(j_i)$ being the priority of task j_i , and a set of resources $R = r_1, \dots, r_k$. Claiming of resources are required to be in LIFO order (for our purpose, we can see this as nested critical sections). Each resource is associated a ceiling value $\pi(r_i)$, given the maximum priority of all tasks in J that requests (claims) the resource. During run-time, the system ceiling (Π) is defined from the maximum ceiling value of all currently claimed resources. Tasks execute, run-to competition (without actively blocking). A task is allowed to start executing only if it has the highest priority of pending tasks, and has a priority higher than the current system ceiling.

Scheduling under SRP has the following key characteristics:

- dead-lock free execution (a task is not permitted to start unless all resources required to finish are available, this is ensured by the scheduling condition),
- bound blocking time, (being, at most, the critical section that a lower priority task, holds a resource with a ceiling value preventing the higher priority task from being executed),
- memory efficiency (all tasks share a common stack), and
- execution/CPU efficiency, (at most one context switch for each task invocation, no yielding/re-scheduling),

A wide variety of analysis techniques has been developed for SRP, including response-time analysis and (maximum) stack memory requirements.

In prior work we have shown how SRP base models can efficiently executed onto bare metal hardware by the RTFM-kernel primitives. The kernel upholds the SRP invariants by efficiently exploiting the underlying interrupt hardware for static priority scheduling and (single unit) resource management.

C. RTFM-core

The RTFM-core language is based on the notions of tasks and resources in correspondence to the Stack Resource Policy (SRP), for a detailed description on the original work on -core we refer the reader to [5] and the upcoming [6]. Here we give a brief overview.

In -core each task is run-to-completion (without any notion of waiting for additional events), and it may *pend* events arbitrarily (this can be seen as requesting asynchronous execution of the corresponding task); each resource may be *claimed* for the duration of a critical section of a task; furthermore, resources can only be claimed in a LIFO (hierarchical) manner, enforcing nested critical sections.

```

Top ::= #> CCode <#
      | ISR Id Int { Stmt }
      | Task Id Int { Stmt }
      | Func Id (CCode) { Stmt }
      | Reset { Stmt }
      | Idle { Stmt }
      | Top Top

Stmt ::= #> CCode <#
       | claim Id { Stmt }
       | pend Id
       | sync Id (CCode)
       | Stmt Stmt

```

Figure 1. Simplified grammar for RTFM-core

With resource constrained systems in mind, the -core language and run-time system implementation imposes further restrictions, namely: events are single unit, i.e., each event can be either pending or not; an event is considered to be consumed when the corresponding task starts executing; resources are single unit, i.e., each resource can either claimed or free; priorities are considered as being static.

In prior work we have shown how such models can be efficiently executed onto bare metal hardware by the RTFM-kernel primitives, [7]

Programs (models) written in -core can be automatically translated into C-code, where invocations of scheduling primitives are generated by the `rtfm-core` compiler, and thus correct by construction w.r.t the task and resource model. Furthermore, the necessary analysis for SRP based scheduling is performed by the `rtfm-core` compiler directly on the given program.

1) *Syntax*: A simplified grammar is depicted in Figure 1. The extended language remains the outset of simplicity, with a clear focus on constructs for concurrency and real-time operations. The `CCode` terminal denotes the presence of embedded C-code, occurring either at the top level, or inside a `ISR/Task/Func/Reset/Idle` statement. (`Func`'s are used merely to facilitate modularization.)

Adding to original -core is the introduction of `Idle`. This allows to discriminate reset statements (executed as a setup procedure before any other tasks are released) to idle statements, executing concurrently to other tasks in the system.

D. Dynamic Semantics

In the following, we give an informal dynamic (run-time) semantics for the -core language, following the notions of *task*'s and *resource*'s.

Each `ISR`, `Task` and `Reset/Idle` definition is bound to a sequence of statements and is referred to as a *task* definition. Execution of a `ISR/Task` (a *task instance*) is triggered by the occurrence of a corresponding *event* and should run-to-completion (i.e terminate). An event can either occur from the environment (typically an interrupt) or be emitted internally (by means of an asynchronous `pend`).

In the following we use `ISR/Task`, *task* and *task instance* interchangeably in case not explicitly referred to as a def-

inition or instance. `Idle` is a (potentially non-terminating) task with the lowest priority in the system, invoked once by the underlying run-time system at startup (after run-to-end execution of the `Reset` task). This allows the user to implement target/application specific power save modes, or background jobs in the `Idle` task.

The `Task`'s are concurrent in our model (and may execute in parallel, besides `Reset` that run before any other task is released). Their associated priorities *may* be used by the underlying scheduler (and corresponding resource and scheduling analysis).

During execution a *task* may request (`claim`) a (single-unit) *resource* for the duration of a *critical section*. Following the grammar, resources will be claimed in a LIFO manner. Functions execute synchronously (`sync`) on behalf of the sender.

III. C-BASED CONTROL FLOW

For the embedded C-code, control flow constructs must be adequately handled in order to ensure compliance to the SRP requirements. In this section we briefly review the control flow constructs of the C-language. RTFM-core embeds arbitrary C-language dialects, we refer the reader to the language definitions for C-89, C99 and C11, for detailed information regarding the specifics for the chosen dialect. The `rtfm-core` compiler is agnostic to the embedded C-code dialect (the embedded C-code is treated literally throughout compilation). The code generation relies merely on C-89 constructs for the translation of -core constructs, and thus to maximise backend compatibility, and compliance to the MISRA subset of C defined to improve robustness and verifiability of C-based software.

A. Single entrance single exit sections

We refer to a section of code having single entry and single exit points as a *basic block* or *plain* section. These can be expressed by the following C-constructs:

- procedures/functions with implicit or explicit end-return
- conditional constructs
 - **if section (else section)** constructs
 - **switch case/default section** constructs
- iterative constructs
 - **for section** constructs
 - **while section** constructs

Conditional and iterative constructs may be nested.

B. Sections with multiple entrance and exits

We refer to a section of code having multiples entry and/or exit points as a set of *basic blocks* or a *broken* section. These can be expressed by the following C-constructs:

- conditional **return** statements
- **break** and **continue** statements
- **goto** statements

IV. CONTROL-FLOW IN RTFM-CORE

As discussed in Section II-B, SRP stipulates that resources should be *requested* and *released* in a LIFO manner. In RTFM-core this is enforced by claiming a (named) resource for a section of code. In effect a `claim R { section }` amounts to resource *request* of R, execution of the *section* and a resource *release* of R. Thus, the LIFO requirement of SRP is upheld as long as the *section* is a basic block (i.e., non-*broken* sequence of instructions). Moreover, a *section* may contain inner *sections* (nested claims). However, the arbitrary embedding of C code may results in *broken* sections, and thus violate the SRP requirement. In previous work, *sections* have been limited to *basic* blocks, in this work we relax this restriction in order to improve flexibility of the programming model, while still remaining correct by construction w.r.t to LIFO nesting requirement of SRP.

In the following we analyse, case by case, the C based control flow of broken sections discussed in Section III-B. We discuss its relation to the RTFM-core language, and propose solutions and their implementation in the `rtfm-core` compiler (in terms of OCaml code).

A. Conditional **return** statements

We have two (related) problems at hand, firstly we need to ensure that claimed resources are released in LIFO manner before returning, and secondly, we need to ensure that the return value is not exposed to race conditions.

To address the first issue, we need to track the occurrence of **return** statements in the `-core` program, record the (nested) resources, and release them in LIFO order.

To address the latter case, we have the option to either require atomic resource release and return functionality, or solve the problem programatically. We opted for the second alternative for several reasons, firstly, atomic operations infers additional blocking, and secondly it requires changes to the run-time architecture (and thus the range of existing implementations).

1) *Approach and -core extension*: We propose to extend the `-core` syntax with the additional keyword **claim_return**, (where the prefix *claim* is chosen to indicatively to its purpose). We propose to prevent (potential) race conditions programatically, by enforcing complying semantics on the return value.

Listing 1 depicts a simple `-core` function `f (int i)`, which exemplifies the **claim_return** conditionally outside claims (line 2, A), inside a single claim (line 4, B) and nested claims (line 6, C), and unconditionally (line 9, D). For the example, the function is invoked from the `Idle` task, triggering the different return conditions.

Listing 2 depicts an excerpt of the generated C-code (the snippet has been post processed for improved readability). The `rtfm-core` compiler specialises function instances (by tracing the call chain), for the purpose of efficient message

buffer management.¹ Line 4 introduces an automatic variable, used for protecting the return value, as used in lines 11 and 18. Lines 4,5 and 25,26 shows returns outside claims (corresponding to cases A and D respectively). Case B (returning from within **claim** R1) is handled by lines 10,...,14, and case C (returning from within the nested claims on R2, R1, R2 being the inner claim) is handled by lines 17, ..., 21. Notice the LIFO order, releasing the inner resource first (line 19).

2) *Implementation in rtfm-core*: Listing 3 depicts an excerpt of the `GenC.ml` that implements the C-code generation for **claim_return** statements in the `rtfm-core` compiler. (Generated C-code comments are omitted for brevity). The input program is parsed into an AST, where a `Return(c, [])` is constructed for each **claim_return** `#> c <#`. Prior to code generation, the AST is analysed for critical sections (Listing 6), such that `cs` in `Return(c, cs)` holds a list of claimed resources for the corresponding **claim_return** in the `-core` program. Line 6, checks if the list is empty (we then have a native C return, as in cases A and D). If not, we check if `c` is empty (Line 7), corresponding to a procedure (**void** function), and in such case generate code to release each claimed resource (Line 8), and return (line 9). If `c` is non-empty (a C-code expression), we generate code to compute and bind the return value to `ret_val` (line 11), release resources (line 12) and return `ret_val` (line 12). This enforces a copying semantics for the return value, and resources can be released safely w.r.t., race conditions on the return value. tiny

```

1  Func int f (int i) {
2    #> if (i == 0) <# claim_return #> 0 <#; // A
3    claim R1 {
4      #> if (i == 0) <# claim_return #> 1 <#; // B
5      claim R2 {
6        #> if (i == 0) <# claim_return #> 2 <#; // C
7      }
8    }
9    claim_return #> i <#; // D
10 }
11
12 Idle {
13   #>
14   printf("f(0)=%d\n", <# sync f(0) #>);
15   printf("f(1)=%d\n", <# sync f(1) #>);
16   printf("f(2)=%d\n", <# sync f(2) #>);
17   printf("f(3)=%d\n", <# sync f(3) #>);
18   <#
19 }

```

Listing 1. Return.core.

B. **break** and **continue** statements

The **break** C statement is used in conjunction with **switch**, and iterative **for**, **while** constructs, in order to force execution to the exit point of the construct. For the iterative constructs, the **continue** C statement forces execution to the entry point of the construct.

To address these issues, we need to track the occurrence of **switch**, **for** and **while** constructs in the `-core` program, record the (nested) resources of **break** and **continue**

¹For this example, identical implementations are rendered for the instances of `f (int i)`, and we depict only the instance `user_idle_f_0 (int i)`. Future work includes detecting and folding identical implementations to reduce code foot-print.

```

1 // Function instance implementation for
2 // Func : int user_idle_f_0
3 int user_idle_f_0(int i) {
4     int ret_val;
5     if (i == 0) // return outside claim (native C return)
6         return 0;
7
8     RTFM_lock(user_idle_nr, R1);
9     if (i == 0) {
10         // code generated to return from within claims : R1
11         ret_val = 1;
12         RTFM_unlock(user_idle_nr, R1);
13         return ret_val;
14     }
15     RTFM_lock(user_idle_nr, R2);
16     if (i == 0) {
17         // code generated to return from within claims : R2, R1
18         ret_val = 2;
19         RTFM_unlock(user_idle_nr, R2);
20         RTFM_unlock(user_idle_nr, R1);
21         return ret_val;
22     }
23     RTFM_unlock(user_idle_nr, R2);
24     RTFM_unlock(user_idle_nr, R1);
25     // return outside claim (native C return)
26     return i;
27 }

```

Listing 2. Excerpt of Return.c.

```

1 | Return (c, cs) ->
2 let unlock r = "RTFM_unlock(" ^ e_path ^ "," ^ r ^ " ^ ";
3 in (
4     match cs with
5     | [] -> "return;" ^ String.trim c ^ ";
6     | _ ->
7         if String.compare "" c == 0 then
8             "(" ^ String.concat nl (List.map unlock cs) ^
9             "return;" ^ nl
10        else
11            "{_ret_val_= " ^ String.trim c ^ "; " ^ nl ^
12            String.concat nl (List.map unlock cs) ^ nl ^
13            "return_ret_val;}" ^ nl )

```

Listing 3. Excerpt of GenC.ml.

statements in such constructs and generate code to release them in LIFO order.

Similarly to the treatment of **claim_return** we opt to solve multiple resource releases in a programatic manner.

1) *Approach and -core extention:* We introduce the (new) keywords **claim_switch**, **claim_for**, **claim_while**, **claim_break** and **claim_continue** to the -core language, and use them to discriminate the handling of resources (in addition to the corresponding C-semantics).

Listing 4 depicts and **switch**-core example, with an excerpt of the rendered code shown in Listing 5. When recording the nesting of case A, **claim_break** (Listing 4, Line 8) it refers solely to the inner claims of the **claim_switch**, i.e., R2. In the generated code (Listing 5) this amounts to the lines 11...15.

For **case B** (Listing 4, Line 10), the **claim_break** is recorded outside any claims within the **claim_switch**, and hence the generated code amounts to a native C-**break** (Listing 5, Lines 19,20).

2) *Implementation in rtfm-core:* The file TaskGenSpec.ml specialises the tasks and functions. The **cs** defines the claim stack (as list of claimed named-resources) for the task/function, while **cs_con** defines the claim stack for the construct at hand (in our case the **claim_switch** amounting to Listing 6 Lines 9...11). Notice, when specialising a Switch (line 11), the **cs_con** is initially set to [] giving us an (initially) empty claim stack. When specialising the **claim_break** (Listing 6 Lines 19,20), the Break is associated to the **cs_con** (claim stack)

of the current construct.

The corresponding code degeneration is depicted in Listing 7 (stripped for brevity from generating comments). For case A, the non-empty **cs** (claim stack) matches the lines 6,...10, while the case B, renders an empty **cs** in line 5.

3) *for and while constructs:* The handling of **for** and **while** constructs follows the same pattern as discussed above.

Listings 8 and 10 depicts **for** and **while** examples with the corresponding generated code shown in Listings 9 and 11 respectively. For cases A, a single resource R2 is claimed within the construct (**claim_for/claim_while**), whereas cases B, showcases the handling of multiple resources (R3, claimed within R2). In both cases the resource R1 is claimed outside the **claim_for/claim_while** constructs, and hence not released by the **inner claim_break/claim_continue** statements.

```

1 Func int f (int i) {
2     #> int j = 0; <#
3     claim R1 {
4         claim_switch (i) { #>
5             case 1: <# // A
6                 claim R2 {
7                     #> j = 1; <#
8                     claim_break;
9                 } #>
10            case 2: j = 2; <# claim_break; #> // B
11            default:
12                j = -1; <#
13            }
14        } #>
15        #> return j; <#
16    }

```

Listing 4. Excerpt of Switch.core.

```

1 int user_idle_f_0(int i) {
2     int ret_val;
3     int j = 0;
4     RTFM_lock(user_idle_nr, R1);
5     switch (i) {
6         case 1:
7             RTFM_lock(user_idle_nr, R2);
8             j = 1;
9             // code generated to break within claims : R2
10            {
11                RTFM_unlock(user_idle_nr, R2);
12                break;
13            }
14            RTFM_unlock(user_idle_nr, R2);
15        case 2:
16            j = 2;
17            // break outside claim (native C break)
18            break;
19        default:
20            j = -1;
21    }
22    RTFM_unlock(user_idle_nr, R1);
23    return j;
24 }

```

Listing 5. Excerpt of Switch.c.

C. goto label statements

The **goto id C** statement is used in conjunction with a corresponding **label id:**. In effect this means that we have arbitrary entry and exit points to sections. Similarly to previous approaches, we opt to solve these problem statically (at compile time) and programatically (i.e., without alternations to the run-time systems).

1) *Approach and -core extention:* We need to record the nesting level for both **goto** statements and **id:** constructs. To this end, we introduce the new -core constructs **claim_goto**

```

1 let spec_of_p topl =
2   let rec stmts cs cs_con i_dl path sal sfl s1 =
3     match s1 with
4     | [] -> []
5     | Claim (r, cs1) :: l ->
6       Claim (r, stmts (r::cs) (r::cs_con) i_dl
7         (path ^ "-" ^ r) [] [] cs1)
8       :: stmts cs cs_con i_dl path sal sfl l
9     | Switch (p, s1) :: l ->
10      Switch (p, stmts cs [] i_dl path [] [] s1)
11      :: stmts cs cs_con i_dl path sal sfl l
12     | For (p, fsl) :: l ->
13      For (p, stmts cs [] i_dl path [] [] fsl)
14      :: stmts cs cs_con i_dl path sal sfl l
15     | While (p, fsl) :: l ->
16      While (p, stmts cs [] i_dl path [] [] fsl)
17      :: stmts cs cs_con i_dl path sal sfl l
18    ...
19  | Return (c, _) :: l ->
20    Return (c, cs) :: stmts cs cs_con i_dl path sal sfl l
21  | Break (c) :: l ->
22    Break (cs_con) :: stmts cs cs_con i_dl path sal sfl l
23  | Continue (c) :: l ->
24    Continue (cs_con) :: stmts cs cs_con i_dl path sal sfl l
25  | Goto (id, _) :: l ->
26    Goto (id, cs) :: stmts cs cs_con i_dl path sal sfl l
27  | Label (id, _) :: l ->
28    Env.env_labels := (id, cs)::Env.env_labels;
29    Label (id, cs) :: stmts cs cs_con i_dl path sal sfl l

```

Listing 6. Excerpt of TaskGenSpec.ml.

```

1 | Break (cs) ->
2   let unlock r = "RTFM_unlock(" ^ e_path ^ ",_" ^ r ^ " );"
3   in (
4     match cs with
5     | [] -> "break;";
6     | _ ->
7       "(" ^ nl ^
8       myconcat nl (List.map unlock cs) ^
9       "break;" ^ nl ^
10      ")" ^ nl )

```

Listing 7. Excerpt of GenC.ml w.r.t Break.

and **claim_label** as show in Listing 12. The example designed to stress the edge cases, a **goto**:

- A from an empty claim stack to a section with a non-empty claim stack ([R2, R1], R2, being the inner claim),
- B from a non-empty claim stack ([R1]) to a section with a non-empty claim stack ([R3, R2, R1], R3, being the innermost claim), where an outer claim stack is partially shared ([R1]),
- C from a non-empty claim stack ([R2, R1]) to a section with a non-empty claim stack ([R2, R1]), where claim stacks are equivalent, and
- D from a non-empty claim stack ([R3, R2, R1]) to a section with a non-empty claim stack ([R2]), where claim stacks are disjoint.

Listing 13 depicts the generated C code:

- A lines 6,...,13 claims the resources of the branch target in order [R1, R2],
- B lines 16,...,23 claimed the additional resources of the branch target in order [R2, R3] (resource R1 already claimed),
- C lines 27,...,32 amounts to a native C **goto** since claim stacks are equivalent, and
- D lines 35,...,45 releases the claims in LIFO order ([R3, R2, R1]) and claims [R2] of the branch target before performing the **goto**.

2) *Implementation in rtfm-core*: Now familiar, file TaskGenSpec.ml (Listing 6 specialises the tasks and functions. Lines 25,26 records the claim stack for each

```

1 Func int f (int i) {
2   #> int j = 0; <#
3   claim R1 {
4     claim_for (int k = 0; k<5; k++) {
5       claim R2 ( #>
6         if (i == 0) <# claim_break; // A
7         claim R3 ( #>
8           if (i == 1) <# claim_continue; // B
9         ) #>;
10        j++; <#
11      }
12    }
13  }
14  #> return j; <#
15 }

```

Listing 8. Excerpt of For.core.

```

1 int user_idle_f_0(int i) {
2   int ret_val;
3   int j = 0;
4   RTFM_lock(user_idle_nr, R1);
5   for (int k = 0; k<5; k++) {
6     RTFM_lock(user_idle_nr, R2);
7     if (i == 0) {
8       // code generated to break within claims : R2
9       RTFM_unlock(user_idle_nr, R2);
10      break;
11    }
12    RTFM_lock(user_idle_nr, R3);
13    if (i == 1) {
14      // code generated to continue within claims : R3, R2
15      RTFM_unlock(user_idle_nr, R3);
16      RTFM_unlock(user_idle_nr, R2);
17      continue;
18    }
19    RTFM_unlock(user_idle_nr, R3);
20    j++;
21    RTFM_unlock(user_idle_nr, R2);
22  }
23  RTFM_unlock(user_idle_nr, R1);
24  return j;
25 }

```

Listing 9. Excerpt of For.c.

claim_goto, whereas lines 27,...,30, records the claim stack for each **claim_label**, and updates Env.env_labels environment, with a binding from the label identifier id to the corresponding claim stack cs.

Listing 14 depicts an excerpt of the code generation (stripped from generating C-code comments). enter_cs looks up the claim stack for the branch target, whereas the function g_l matches goto to label resources for the claim stack. When invoked (line 11) the arguments (claim stacks) are passed in reverred order. Hence the matching starts from the outermost claimed resource. A case by case study of the function yields:

- A line 4 matches directly (the claim_goto has an empty stack) and g_l returns ([], [R1, R2]), hence r1 = [] and no resources are released (line 14), and c1 = [R1,R2] (line 15) renders the claims of lines 10,11 in Listing 13,
- B line 8 first matches the outermost common resource R1, then line 4 matches and g_l returns ([], [R2, R3]), which renders the claims of lines 20,21 in Listing 13,
- C line 8 matches (recursively) until line 4 matches and g_l returns ([], []), thus neither releases, nor claims are produced, and
- D line 9 matches directly (R1 ≠ R2) and g_l returns ([R3, R2, R1], [R2]), which renders (line 14) the resource releases (lines 40,...,42, Listing 13) and the resource claim (line 43, Listing 13).

(To make the elaboration complete we should also stress line

```

1  Func int f (int i) {
2    #> int j = 0; <#
3    claim R1 {
4      #> int k = 0; <#
5      claim_while (k < 5) {
6        #> k++; <#
7        claim R2 { #>
8          if (i == 0) <# claim_break; // A
9          claim R3 { #>
10             if (i == 1) <# claim_continue; // B
11             } #>;
12             j++; <#
13             }
14         }
15     }
16     #> return j; <#
17 }

```

Listing 10. Excerpt of While.core.

```

1  int user_idle_f_0(int i) {
2    int ret_val;
3    int j = 0;
4    RTFM_lock(user_idle_nr, R1);
5    int k = 0;
6    while (k < 5) {
7      k++;
8      RTFM_lock(user_idle_nr, R2);
9      if (i == 0) {
10         // code generated to break within claims : R2
11         RTFM_unlock(user_idle_nr, R2);
12         break;
13     }
14     RTFM_lock(user_idle_nr, R3);
15     if (i == 1) {
16         // code generated to continue within claims : R3, R2
17         RTFM_unlock(user_idle_nr, R3);
18         RTFM_unlock(user_idle_nr, R2);
19         continue;
20     }
21     RTFM_unlock(user_idle_nr, R3);
22     j++;
23     RTFM_unlock(user_idle_nr, R2);
24 }
25 RTFM_unlock(user_idle_nr, R1);
26 return j;
27 }

```

Listing 11. Excerpt of While.c.

5, but that can be seen as a special case of line 9, when the branch target is outside any claims, and thus omitted for the presentation.)

Code generation for labels is straightforward, and amounts merely to generating code for a native C label (lines 18,19).

D. Completeness and optimisation

For the analysis and code generation, we do not make any attempts to analyse the embedded C code (reachability, etc.), and hence we do not perform any optimisation w.r.t control flow.

```

1  Func int f (int i) {
2    #> int j = 0;
3    if (i == 1) <# claim_goto l1; // A
4    claim R1 { #>
5      j++;
6      if (i == 2) <# claim_goto l2; // B
7      claim R2 {
8        claim_label l1: #>
9        j++;
10       if (i == 3) <# claim_goto l3; #> // C
11       j++; <#
12       claim R3 {
13         claim_label l2: #>
14         if (i == 4) <# claim_goto l4; // D
15         #> j++; <#
16       }
17       #> j++; <#
18       claim_label l3:
19     }
20 }
21 claim R2 {
22   claim_label l4:
23 }
24 #> return j; <#
25 }

```

Listing 12. Excerpt of Goto.core.

```

1  // Function instance implementation for
2  // Func : int user_idle_f_0
3  int user_idle_f_0(int i){
4    int ret_val;
5    int j = 0;
6    if (i == 1) {
7      // goto label l1
8      // from claims [], to claims [R2, R1]
9      // -> exit claims [], enter claims [R1, R2]
10     RTFM_lock(user_idle_nr, R1);
11     RTFM_lock(user_idle_nr, R2);
12     goto l1; // native C goto
13 }
14 RTFM_lock(user_idle_nr, R1);
15 j++;
16 if (i == 2) {
17   // goto label l2
18   // from claims [R1], to claims [R3, R2, R1]
19   // -> exit claims [], enter claims [R2, R3]
20   RTFM_lock(user_idle_nr, R2);
21   RTFM_lock(user_idle_nr, R3);
22   goto l2; // native C goto
23 }
24 RTFM_lock(user_idle_nr, R2);
25 l1: // native C label at claims [R2, R1]
26 j++;
27 if (i == 3) {
28   // goto label l3
29   // from claims [R2, R1], to claims [R2, R1]
30   // -> exit claims [], enter claims []
31   goto l3; // native C goto
32 }
33 j++;RTFM_lock(user_idle_nr, R3);
34 l2: // native C label at claims [R3, R2, R1]
35 if (i == 4) {
36   // goto label l4
37   // from claims [R3, R2, R1], to claims [R2]
38   // -> exit claims [R3, R2, R1],
39   // enter claims [R2]
40   RTFM_unlock(user_idle_nr, R3);
41   RTFM_unlock(user_idle_nr, R2);
42   RTFM_unlock(user_idle_nr, R1);
43   RTFM_lock(user_idle_nr, R2);
44   goto l4; // native C goto
45 }
46 j++;
47 RTFM_unlock(user_idle_nr, R3);
48 j++;
49 l3: // native C label at claims [R2, R1]
50 RTFM_unlock(user_idle_nr, R2);
51 RTFM_unlock(user_idle_nr, R1);
52 RTFM_lock(user_idle_nr, R2);
53 l4: // native C label at claims [R2]
54 RTFM_unlock(user_idle_nr, R2);
55 return j;
56 }

```

Listing 13. Excerpt of Goto.c.

```

1  | Goto (id, g_cs) ->
2  let l_cs = (Env.lookup_label id) in
3  let rec g_l = function
4  | ([], l1) -> ([], l1)
5  | (gl, []) -> (List.rev gl, [])
6  | (g::gl, l::l1) ->
7    match (String.compare g l) with
8    | 0 -> g_l (gl, l1)
9    | _ -> (List.rev (g::gl), (l::l1))
10
11 in
12 let (r1, c1) = g_l (List.rev g_cs, List.rev l_cs)
13 in
14 "_{ " ^ nl ^
15 myconcat nl (List.map unlock r1) ^
16 myconcat nl (List.map lock c1) ^
17 "}" ^ nl
18 | Label (l, enter_cs) ->
19 1 ^ " " ^ " " ^ nl

```

Listing 14. Excerpt of GenC.ml w.r.t goto.

V. IMPLEMENTATION

The proposed extensions to the -core language has been implemented in the *rtfm-core* compiler. Of most interest for this presentation, the OCaml files `AST.ml` (defining the internal program representation for the parsing, `Parser.ml`) of the input -core program, and the file `SpecAST.ml` (defining the data structures for the specialisation, `Parser.ml`), and `GenC.ml` (producing, the C-code compiler output).

The presented extensions works without any alternations to the run-time systems, and preserves the race and deadlock free invariants (the latter applies to execution by the RTFM-kernel).

VI. RELATED AND FUTURE WORK

In the context of concurrency support for C-based programming we find a wide range of light weight libraries, kernels and operating systems, e.g., Chibios [8], FreeRTOS [9] and RIOT [10], which in common provide thread abstractions. Other lightweight alternatives targeting Internet of Things are e.g., in the *makers community* the Aurdiono OS (which is essentially a library of commodity components with a focus of simplicity of use rather than a fully fledged scheduler), Contiki [11], providing a thread abstraction proto-threads (essentially C-macros implying cleverly nested **switch** statements mimicking threads without priorities), and TinyOS (which provides a two tier scheduling model with a preemptive layer (essentially interrupts) for which *atomic* critical sections are provided to protect shared resources).

RTFM-core provides a programming model with language constructs based on fully pre-emptive and time constrained tasks, which shared resources protected by (named) critical sections. In common, none of the aforementioned approaches provides such a uniform programming model for concurrent real-time software. In contrast they require the programmer to manually encode sought timing properties in the model/API or even go to the length stepping outside the model (taking over the underlying bare-metal interrupts), as in Aurdiono and Contiki. W.r.t. context handling, RTFM-core (through the RTFM-kernel) offers single stack execution, similarly to Aurdiono OS, Contiki and TinyOS, whereas the other mentioned approaches essentially deploy traditional context handling for thread based systems. W.r.t. programming model, TinyOS goes on the length of language constructs (in terms of the C which offers a two tier execution model with preemptive interrupts (events) and non-preemptive background tasks).

In the literature we find other work, extending the C language with constructs of concurrency. The most prominent is Concurrent C [12], [13]. In their approach, the user has to take explicit responsibility for creating and maintaining processes, buffers etc., while in the RTFM-core language this is automated by the compiler. Our more restricted model facilitates analysis and allows correct by construction implementations.

The RTFM-core language and compiler is under active development. We are currently investigating the addition of tasks with parameters (arguments for the task instance execution), resembling traditional message passing. Moreover, we are investigating the addition of timing semantics, where task priorities are derived from the timing requirements (deadlines) for the task instance. Additionally, asynchronous invocation of tasks may be postponed in relation to the sender's baseline (release time). This will allow timing requirements to be expressed directly in the RTFM-core language and allow for compile time analysis of timing properties and required buffer space for outstanding messages. With this at hand, we project automatic generation of low-level timer implementations, and correct by construction timing properties to be feasible. We refer the reader to [14] for up to date information on RTFM-lang developments.

VII. CONCLUSIONS

Lock based resource protection is in general difficult for the programmer. The RTFM-core language provides correct by construction LIFO locking patterns through the introduction of critical sections directly in the language. In this paper we have extended the -core language with a set of constructs allowing for C-based control flow, providing correct by construction lock management. We have shown that the LIFO order is preserved, thus the requirements for SRP based analysis is maintained. Moreover, an experimental implementation in the `rtfm-core` compiler has been presented, and we have demonstrated its feasibility through a set of representative running examples, and shown in detail how C-based control flow is handled through compilation and managed during run-time execution by the RTFM-kernel.

REFERENCES

- [1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: <http://dx.doi.org/10.1109/MC.2006.180>
- [2] P. Lindgren, D. Pereira, J. Eriksson, M. Lindner, and L. M. Pinho, "RTFM-lang Static Semantics for Systems with Mixed Criticality," in *Ada-Europe 2014: 19th International Conference on Reliable Software Technologies*, 2014.
- [3] P. Lindgren, M. Lindner, and et.al, "Rtfm-core: Course in compiler construction," in *ESWEEK/WESE 2014*, 2014.
- [4] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec. 1990, pp. 191–200.
- [5] P. Lindgren, M. Lindner, and et.al, "RTFM-core: Language and Implementation," in *ESWEEK/CPSArch 2014*, 2014.
- [6] P. Lindgren, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho, "RTFM-core: Language and Implementation," in *Accepted to: 10th IEEE Conference on Industrial Electronics and Applications (ICIEA 2015)*, 2015.
- [7] J. Eriksson, F. Haggstrom, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives," in *SIES. IEEE*, 2013, pp. 110–113. [Online]. Available: <http://dblp.uni-trier.de/db/conf/sies/sies2013.html>
- [8] Chibios/RT. (webpage) Last accessed 2014-07-28. [Online]. Available: <http://www.chibios.org>
- [9] FreeRTOS. (webpage) Last accessed 2014-07-28. [Online]. Available: <http://www.freertos.org>
- [10] RIOT. (webpage) Last accessed 2014-07-28. [Online]. Available: <http://riot-os.org>
- [11] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004. [Online]. Available: <http://dunkels.com/adam/dunkels04contiki.pdf>
- [12] N. H. Gehani and W. D. Roome, "Concurrent c," *SOFTWARE-PRACTICE AND EXPERIENCE*, vol. 16, pp. 821–844.
- [13] N. H. Gehani and Roome, "Implementing concurrent c," *Software: Practice and Experience*, vol. 22, no. 3, pp. 265–285, 1992.
- [14] RTFM: Real-Time For the Masses. (webpage) Last accessed 2014-09-19. [Online]. Available: <http://www.rtfm-lang.org>