Implementation of SRP-DM Scheduling for Embedded Real-Time Software

Johan Eriksson*, Simon Aittamaa*, Jimmie Wiklander*, Pawel Pietrzak* and Per Lindgren*
*EISLAB, Luleå University of Technology, 971 87 Luleå
Email: {Johan.Eriksson/Simon.Aittamaa/Jimmie.Wiklander/Pawel.Pietrzak/Per.Lindgren}@ltu.se

Abstract—Model and component based design is an established means for the development of large software systems, and is starting to get momentum in the realm of embedded software development. In case of safety critical (dependable) systems it is crucial that the underlying model and its realization captures the requirements on the timely behavior of the system, and that these requirements can be preserved and validated throughout the design process (from specification to actual code execution). To this end, we base the presented work on the notion of Concurrent Reactive Objects (CRO) and their abstraction into Reactive Components.

In many cases, the execution platform puts firm resource limitations on available memory and speed of computations that must be taken into consideration for the validation of the system.

In this paper, we focus on code synthesis from the model, and show how specified timing requirements are preserved and translated into scheduling information. In particular, we present how ceiling levels for Stack Resources Policy (SRP) scheduling and analysis can be extracted from the model. Additionally, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. Moreover, the design of a micro-kernel supporting cooperative hardware- and software-scheduling of CRO based systems under Deadline Monotonic SRP is presented.

I. INTRODUCTION

Model and Component Based Design (CBD) has over the years proven to be an effective means for the development of large software systems. Key drivers behind are to increase the efficiency of the design process (mainly through re-use of components) and to improve the product quality (mainly by facilitating the validation process through allowing for separate verification of components).

With the ever increasing complexity of embedded systems, CBD of embedded software is starting to gain momentum.

In case of embedded, safety critical systems, it is crucial that the underlying model and its realization can capture the requirements on the timely behavior of the system, in terms of both external and internal interactions, and that these requirements can be preserved and validated throughout the design process (from specification to actual code execution). In many cases, the execution platform puts firm resource limitations on available memory and speed of computations, that must be taken into consideration for the validation of the system. Thus, a straight forward transition of traditional CBD models [1], [2] and tools does not suffice.

We undertake the component model and accompanying design methodology presented in our earlier work [3]. With the outset from a reactive system view, the behavior of the system can observed as its output to incoming events. In general, system output also depends on previous events. Thus, embedded systems of scale are stateful. To deal with complexity and allow for CBD of such systems, we partition state and functionality into a hierarchy of Concurrent Reactive Components (CRC). Components are specified in terms of Concurrent Reactive Objects (CRO) instances and component instances [4], CRO allows the intended behavior of the system to be expressed in terms of Time-Constrained Reactions (TCR) [5]. A CRO instance is either implemented in software (e.g., synthesized from the CRO model) or implemented by the system’s environment. This allows incorporating hardware interactions and legacy code (typically external software libraries) in the model, as long as their interface is compliant with the CRO model.

In this paper, we focus on code synthesis of CRO models and extraction of information for scheduling (during run-time) as well as offline schedulability analysis. To this end, we show how specified timing requirements (in the CRO model) are preserved and translated into scheduling information, specifically resource ceilings and priority levels for Stack Resource Policy (SRP) scheduling. Moreover, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. Additionally we present the design of a micro-kernel supporting SRP Deadline Monotonic (SRP-DM) scheduling of CRO based systems, exploiting the interrupt hardware for efficient scheduling.

Section II gives the necessary background. We detail the underlying CRO model (and its abstraction to the CRC model) and briefly recapture the notions and key features of SRP. In section III, we give an informal mapping from the undertaken CRO model to the notions of SRP. In Section VI, we present an example system (a process controller), and show how system state, functionality and temporal properties are captured and abstracted in terms of the CRC model. In Section V, we propose a method for code synthesis of CRO models, and show how the timing requirements from the specification is preserved. In Section IV, we propose an algorithm that from a CRO model extracts resource ceilings and priorities for SRP-DM based scheduling. The algorithm is exemplified on the process controller, showing how timing requirements from the model specification is translated into resource ceilings and priorities for the scheduler. Furthermore, in Section VII, we present the design of an efficient SRP-DM kernel, and
demonstrate how the derived resource ceilings and priorities are utilized at run-time. The paper is concluded in Section VIII, where we summarize the presented proposals and results, and give directions for future research in the field.

II. BACKGROUND

A. Underlying Model

1) Concurrent Reactive Object Model: The concurrent reactive object (CRO) model is the execution and concurrency model of the Timber programming language, a general-purpose object-oriented language that primarily targets real-time systems [6]–[8]. A subset of C, TinyTimber [9], implements the core features of Timber and uses CRO as its execution model.

In this section we briefly describe the main features of the CRO model and its abstraction to Concurrent Reactive Components, and discuss its implementation. We present an informal mapping from the CRO model to the notions of SRP. We focus on reactivity; object-orientation with complete state encapsulation; object-level concurrency with message passing between objects, the ability to specify timing behavior of a system, and the abstraction to components.

2) Reactivity: Reactivity is the defining property of the CRO model, which makes it particularly suitable for embedded system, since functionality of most, if not all, embedded systems can be expressed in terms of reactions to external stimuli and timer events. A reactive system can be described as follows: initially the system is idle, an external stimulus (originating in the system’s environment) or a timer event triggers a burst of activity, and eventually the system returns to the idle state. A reactive object is either actively executing a method in response to an external stimulus or a message from another object, or passively maintaining its state. Since initially the system is idle, some external stimulus is needed to trigger activity in the system.

3) Objects and state encapsulation: The CRO model specifies that all system state is encapsulated in objects $O_1, \ldots, O_n$. Each object has a number of methods, and the encapsulated state is only accessible from the object’s methods. This is also known as a complete state encapsulation. A name of a method $m$ can be fully expanded as $O_i : m$, where $O_i$ is the object of the method. Methods of two objects can be executed concurrently, but each method is granted an exclusive access to its object’s state, so only one method of an object can be active at any given time. Coupled with a complete state encapsulation, this provides a mechanism for guaranteeing state consistency under concurrent execution. The source of concurrency in a system can either be two (or more) external stimuli that are handled by different objects, or an asynchronous message sent from one object to another (more about message passing below).

To ensure that execution is reactive in its nature, each method must follow run-to-end semantics [10], i.e., it is not allowed to block execution awaiting external stimulus or a message. An example of this would be an object representing a queue: if a dequeue method is invoked on an empty queue, it is not allowed to wait until data becomes available, but it must instead return a result indicating that the queue is empty.

4) Message passing and specification of timing behavior: In the CRO model objects communicate by passing messages. Each message specifies a recipient object ($O$) and a method ($m$) of this object that will be invoked. A message is either synchronous ($\text{SYNC}(O,m)$) or asynchronous ($\text{ASYNC}(O,m)$). The sender of a synchronous message blocks waiting for the invoked method to complete (with a possible result), while the sender of an asynchronous message can continue execution concurrently with the invoked method. Thus, asynchronous messages introduce concurrency into the system. An asynchronous message can also be delayed by a specific amount of time.

Timing behavior of a system can be specified by defining a baseline and a deadline for an asynchronous message (a synchronous message always inherits the timing specification of the sender). The baseline specifies the earliest point in time when a message becomes eligible for execution, which for an external stimulus corresponds to its “arrival time” and for a message sent from one object to another is defined directly in the code. If the defined baseline is in the future, this corresponds to delaying the delivery of the message. The deadline specifies the latest point in time when a message must complete execution, which is always defined relative to the baseline. Together, baseline and deadline form a permissible window of execution for a message (see Figure 1). Whenever we talk about timing behavior of asynchronous messages, we shall extend the notation to $\text{ASYNC}(O, m, B , D)$, where $B$ and $D$ are respectively a baseline and a relative deadline of the message. As mentioned above, synchronous messages always inherit their timing specification from the initial asynchronous message.

Concurrent reactive objects can be used to model the system itself and its interaction with its environment (e.g., via sensors, buttons, keyboards, displays). Events in the physical world (such as pushing a button) results in an asynchronous message being sent to a handler object, and system output (e.g., flashing an LED) is represented as messages sent from an object to the environment.

5) Implementation of CRO: The implementation of an object instance can be either in software (e.g., synthesized from the class definition in the Timber language, or from a TinyTimber definition in C) or provided by the environment. This allows incorporating hardware interactions and legacy code (typically external software libraries) in the model, as
long as their interface is compliant with the concurrent reactive object model.

6) Concurrent Reactive Components: To deal with complexity, system state and functionality is partitioned into a hierarchy of concurrent reactive components (CRCs) encapsulating component instances and CRO instances (see [11]). Thus, interaction in between components will always be grounded by interaction in between objects (belonging to different components). This allows us to apply a consistent modeling for component interaction, component implementation and interaction with the environment (since a CRC is simply a collection of one or more CROs).

B. Stack Resource Policy

Stack resource policy (SRP) is a policy for scheduling real-time tasks with shared resources that permits tasks with different priorities to share a single run-time stack [12]–[14]. SRP applies directly to scheduling policies with dynamic and static priority, including e.g., Earliest Deadline First (EDF) which is used by current Timber and TinyTimber kernels, and Deadline Monotonic (DM) for which interrupt hardware of commonplace platforms can be utilized efficiently. SRP scheduling offers a number of advantages, mainly deadlock numbers. For our case, with the DM scheduling, we can assign \( \pi(M) = p(M) \) for every \( M \) (since this assignment satisfies the condition for preemption levels, see (P1) in [12]).

This translation into notions of SRP is possible, since methods in the CRO model are run-to-end (blocking for future events to the system is prohibited), thus the execution of a message method can be seen as corresponding to the execution of a job instance.

A. resource ceilings

Assume a message \( M_1 = \text{SYNC}(O_1, m_1, B_1, D_1) \) (or \( M_1 = \text{ASYNC}(O_1, m_1, B_1, D_1) \)). If the execution of \( m_1 \) can give rise to sending a synchronous message \( M_2 = \text{SYNC}(O_2, m_2, B_2, D_2) \) then we write \( M_1 \rightarrow M_2 \). Let \( \rightarrow \) be a transitive closure of \( \rightarrow \). The initial message in a path defined by \( \rightarrow \) may be synchronous or asynchronous, the subsequent messages must be synchronous. The set of resources (objects) potentially requested by a message \( M_0 = \text{ASYNC}(O_0, m_0, B_0, D_0) \) is defined as

\[
\text{Objs}(M_0) = \{O_0\} \cup \{O \mid M_0 \rightarrow \text{SYNC}(O, m, B, D)\}
\]

We also say that \( M \) can lock objects \( \text{Objs}(M) \). A current resource ceiling \([O]\) is defined as

\[
\mathcal{I}^O = \max\{\pi(M) \mid M \in \mathcal{M}, O \in \text{Objs}(M)\}
\]

where \( \mathcal{M} \) stands for all messages in the system. Note that \( [O] \) can be computed statically (if \( \mathcal{M} \) is statically known). An algorithm for computing \([O]\) is discussed in Section IV. The current system ceiling \( \Pi \) is

\[
\Pi = \max\{[0] \} \cup \{ [O] \mid O \text{ is locked} \}
\]

The SRP states (cf. [12]) that a message \( M = \text{ASYNC}(O, m, B, D) \) is blocked from starting execution until \( \pi(M) > \Pi \). In addition to that, in order to be scheduled for execution \( M \) must have highest priority of all jobs, which in the case of DM follows directly from \( \pi(M) > \Pi \).

The derived resource ceilings, along with the base and deadlines of the messages provide sufficient information for the run-time scheduling under SRP, and will ensure that the system passing the analysis will be deadlock-free during execution.

IV. RESOURCE CEILING AND PRIORITY EXTRACTION FOR CRO

In this section we give algorithms for calculating resource ceilings, interarrival time (period) and offsets of jobs (asynchronous messages).

We define the message passing graph of the program as

\[
G = (S, A, N)
\]

where

\[
N = \{m_1, \ldots, m_{\text{nnmeth}}\} \quad (m_i \text{ denotes a method})
\]

\[
S = \left\{ \frac{1}{n_{\text{async}}}, \ldots, \frac{1}{n_{\text{async}}} \right\} \subseteq N \times N \quad (\text{SYNC}(\ldots) \text{ messages}.)
\]

\[
A = \left\{ \frac{1}{n_{\text{async}}} \rightarrow \frac{1}{n_{\text{async}}} \right\} \subseteq N \times N \quad (\text{ASYNC}(\ldots) \text{ messages}.)
\]
**m** represents a method, each *m* has one unique receiving object, represented as *O*(m) (i.e. one method cannot have two different receiving objects).

*S* and *A* is not initially known. Defining *κ*(*m*<sub>i</sub>) as the set synchronous messages that may be sent from a method *m*<sub>i</sub> and *α*(*m*<sub>i</sub>) as the set of asynchronous messages (that may be sent), then *S* and *A* can be calculated as

\[
S = \bigcup_{k=1}^{\text{meth}} \kappa(m_k)
\]

\[
A = \bigcup_{i=1}^{\text{meth}} \alpha(m_i)
\]

For a CRO program to be valid, *S* must be acyclic (otherwise the program may contain deadlocks).

Let *β*(*j* → *) denote the baseline and *δ*(*j* → *) the deadline of an asynchronous message *j* → *. If invoking *m*<sub>i</sub> may result in more than one asynchronous message to *m*<sub>j</sub>, then *α*(*m*<sub>i</sub>) will return a message (*m*<sub>i</sub>, *m*<sub>j</sub>) with the shortest baseline and deadline (baseline and deadline may come from two different messages). The resource ceiling of an object can then be calculated as

\[
[O_i] = \min(C \bigcup \mathcal{V})
\]

\[
\mathcal{V} = \{\delta(\rightarrow*) | \rightarrow* = (m_j, m_i) \in A, O(m_i) = O_i\}
\]

\[
C = \{\delta(\rightarrow*) | \rightarrow* = (m_j, m_k) \in A, (m_k, m_i) \in \delta^*, O(m_i) = O_i\}
\]

where \(\delta^*\) denotes the transitive closure of \(\delta\), \(\mathcal{V}\) denotes the set of asynchronous messages to any method \(m_i\) where \(O(m_i) = O_i\), and *C* the set of asynchronous messages to any method that may (transitively) send a synchronous message to \(m_l\) where \(O(m_l) = O_i\).

Additionally, for a program to be analysed for periods and offsets, the following must hold:
- Any Strongly connected component (SCC) of the graph *G* may not be reachable from another SCC of *G*.
- Any SCC of *G* may only contain one cycle.
- Multiple sync/async from *m*<sub>i</sub> to *m*<sub>j</sub> is not allowed.

Then we can define \(\Gamma\) as a function that calculates the period of any given \(\rightarrow^*\).

\[
\Gamma(\rightarrow^*) = \sum_j \beta(\rightarrow^*) \quad \text{for all } j \text{ part of a SCC that can reach } \rightarrow^*.
\]

For calculating the offset for a \(\rightarrow^*\) relative to method \(m_j\) we define a function \(\Delta(m_j, \rightarrow^*)\) that will return a set of all possible offsets. To calculate this function we must first find all paths from \(m_j\) to (the called method of) \(\rightarrow^*\), for each path we calculate: \(\sum_k \beta(\rightarrow^*)\) for all \(k\) part of the path. The set of results of this calculation is the result of the function \(\Delta\).

### V. Code Synthesis of Framework

The code synthesis framework is depicted in Figure 2. The Reko IDE [11] enables design of CRO models. It provides a GUI where the user can create, browse, and edit the CRO models which are represented graphically in the IDE. The illustrations of the example system presented in Section VI are screenshots from the Reko IDE.

Code generation is an integrated part of the Reko IDE, and is covered in section.

#### A. XML Format

A model is stored in as an XML file that captures both system structure and implementation. For a description of the internal format see [11].

#### B. Requirements on the methods

All methods are written in the C language. However, the C language allows us to specify behavior not allowed by the CRO model (such as entering an infinite loop, waiting for input, etc.). Thus, in order for a method written in C to be a valid CRO method it must comply with the following rules, it must

1. be run-to-end (complete execution within a finite amount of time).
2. not access any global memory (state) outside of its object, and
3. not invoke a method of another object directly (without using the ASYNC or SYNC primitive).

Compliance with two and three are partly enforced by name scoping, i.e. the framework will generate local names for names in the current scope (method). However, the system designer can still force an incorrect behavior by directly calling methods (of other objects) or accessing global memory (e.g., using pointers). Strictly enforcing two and three would require implementing a parser for a subset of the C language (disallowing pointer arithmetic, extern keyword, etc.). The first rules is currently not enforced in any way and compliance must be ensured by the system designer. One approach to enforcing the first rule is to perform worst-case execution time analysis [18] on the methods, but this is outside the scope of this paper.

Below is an example of emitted C code, showing local defines for names in the current scope.

```c
// local name bindings
#define get_feedback .......
#define control_out .......
#define process .....;

// method implementation
int controller_process(OBJ* self, int arg){
    int fb = SYNC(get_feedback, 0);
    // Controller state update etc, eg:
    self->state.out = 10;
    SYNC(control_out, self->state.out);
    ASYNC(process, MS(10), MS(1), 0);
} #undef get_feedback
#undef control_out
#undef process
```

#### C. Code Synthesis

The CRC model contains definitions, instances, methods, and states. From this C functions and object definition structures (C typedefs) can be emitted. Additional information required to compile (using a C-compiler) the system into an executable binary is:
- A static object structure (see below).
- Defines for preemption levels and resource ceilings. (covered in: IV and VI-A)
- A Kernel (VII)
Fig. 2. The code synthesis framework used for translating the CRC model into an executable.

Fig. 4. The internals of the system component definition (object instances in yellow and component instances in blue) with arrows indicating the message paths.

1) Static object structure: Is generated by transforming the CRC model to a model consisting only of object instances (CRO instances). From this model it is possible to generate one C-struct containing all instances in the system. The timing specification of the model is preserved during code synthesis and later used by the run-time kernel, (For reference see the generated C-function above):

VI. EXAMPLE SYSTEM

The component model rely on message passing between components/objects and messages can only be sent between the provided/required interface of an component/object. Every component/object can define a provided and/or a required interface. The provided interface defines ports that can receive messages from other ports. For an object, a port corresponds to a method within the object and, for a component, a port corresponds to a method of an object defined at some level inside the component (components can be hierarchical). The required interface defines ports used for sending messages to the port of an other component/object. However, this require that the sending port is connected to (has a reference to) a receiving port. Let us now present a small controller system and show how it is abstracted in terms of the CRC model.

At top level in the model we have two component definitions (app and system) and four object definitions (env, adapter, controller, and logger), see Figure 3. The env definition encapsulates specific hardware functionality and functions as a gateway between the app and the environment. It has a provided interface with two ports (get_ad, set_pwm) and a required interface with three ports (reset, int1, int2). The app component consists of a single instance of adapter, controller, and logger definition respectively, see Figure 5. The env interrupts (reset, int1, int2) are passed to the app via its provided interface (start, inc, dec). Internally, these are connected to the provided interface of the adapter. The role of the adapter is to forward the interrupts from the environment to the controller and logger in an application specific format (e.g. both inc and dec is forwarded to the controller’s port setpoint but with different arguments that will result in an increase/decrease of the setpoint). The controller functions as a simple feedback controller attempting to minimize the error (i.e. the difference between the measured process variable and the desired setpoint), by adjusting the control signal (control_out). The controller acquires the process variable through the get_feedback port of its required interface. It is connected to get_ad of the env via the required interface port get_val of the app. In a similar manner the controller’s control_out interface port is connected to set_pwm of the env.

The system component consists of a single instance of app and env respectively, see Figure 4. The system component definition is selected as the root for instantiation of the CRC model.

Each method (of a object definition) is implemented in C-code and message passing is done either asynchronously using the ASYNC primitive, or synchronously using the SYNC primitive (see Section II). This is used to specify the timing behavior of the system, e.g. the controller defines the method process:

```c
int fb = SYNC(get_feedback, NO_ARG);
//Controller state update etc
SYNC(control_out, outval);
ASYNC(process, MS(10), MS(1), NO_ARG);
```

The process periodically acquire feedback values and writes control values as specified by the ASYNC primitive (with a period of 10 milliseconds and a deadline of 1 millisecond). Since baseline and deadline of synchronous messages are always inherited, there is no need to supply them when using the SYNC primitive in the code.
A. SRP Levels for the Example

Auto-generated output from the analysis for the given example. Shows all jobs with corresponding calltrees and deadlines (preemption levels is statically assigned according to deadline). Also all acquired resources and corresponding resource ceilings (rc).

```
#Starting points:
JOBREQUEST, dl: 10000
- entry1 rc: 15 entry1 [envinst]
  --start rc: 15 adapter_start [adapterinst]

JOBREQUEST, dl: 50
- entry2 rc: 15 entry2 [envinst]
  --inc rc: 15 adapter_inc [adapterinst]
    ---set_bor rc: 15 controller_set_bor [controllerinst]

JOBREQUEST, dl: 50
- entry3 rc: 15 entry3 [envinst]
  --dec rc: 15 adapter_dec [adapterinst]
    ---set_bor rc: 15 controller_set_bor [controllerinst]

#Detected job requests (internal ASYNCS):
JOBREQUEST, dl: 10000
- init rc: 20 logger_init [loginst]

JOBREQUEST, dl: 10000
- init rc: 15 controller_init [controllerinst]

JOBREQUEST, dl: 20
- log rc: 20 logger_log [loginst]
  --get_ad rc: 15 env_get_ad [envinst]

JOBREQUEST, dl: 15
- process rc: 15 controller_process [controllerinst]
  --get_ad rc: 15 env_get_ad [envinst]
  --set_pwm rc: 15 env_set_pwm [envinst]
```

Here the resource ceiling is given by its deadline, this is not suitable for scheduling since interrupt hardware typically expects priorities given as integers \([0,1,2,...]\) where 0 is most urgent. So all resource ceilings must be sorted and numbered, for this example we generate the following defines:

```
#define __[envinst]_rc 0
#define __[controllerinst]_rc 0
#define __[adapterinst]_rc 0
#define __[loginst]_rc 1
```

This means that [loginst] will be allowed to be pre-empted by any of the other job’s, but no other preemption is possible (preemption is not allowed between job’s having the same resource ceiling). In this example, all interrupts are messages to the same object ([envinst]), thus all other objects in the transitive closure of the highest priority message (to [envinst]) will have a resource ceiling equal to the preemption level of the highest priority message (i.e., 0). However, interrupt priorities are equal to corresponding message priority. Calculation of DM priorities is trivial and emitted in a similar fashion as resource ceilings (i.e., as defines). Note that names such as [loginst] are auto-generated by the compiler, but renamed in the paper for clarity.

VII. KERNEL DESIGN FOR EFFICIENT SRP DM SCHEDULING OF CRO

The goal of the kernel design is to provide an efficient and predictable implementation of the CRO semantics (see Section II-A). In order to achieve this we have decided to use the deadline monotonic (DM) scheduling policy with stack resource policy (SRP). The benefits of SRP are well known (see Section II-B), and DM was primarily chosen to allow for an efficient usage of the priority based interrupt hardware of the target architecture (i.e. Cortex-M3 [19]), more on this in Section VII-C. The scheduler requires that the priorities and preemption levels of messages, and resource ceilings of objects are known. In our case, these are automatically generated from timing specifications in the model during code synthesis (see Section V).

A. Message Passing

The kernel must support two types of messages, asynchronous and synchronous. An asynchronous messages must
be queued until the time it becomes eligible for execution (see permissible window of execution, Figure 1). Queued messages are stored in the timer-queue (sorted by ascending baseline), once the baseline of a message is passed (i.e., the baseline expires) the message is transferred to the active-queue (sorted by descending priority), and considered when scheduling decisions are made (see Section VII-B). The transfer of messages from the timer-queue to the active-queue is initiated by a timer interrupt, i.e. a hardware timer is configured to generate an interrupt when the earliest baseline in the timer-queue expires. When an asynchronous message is scheduled for execution the recipient object-resource (see Section III) is requested, the method is invoked and (when it returns) the object-resource is released. A synchronous message is executed similarly (i.e., object-resource requested, method invoked, and object-resource released).

B. Scheduling

The scheduler only considers messages that are in the active queue, and it is invoked when either a baseline expires (timer interrupt) or the system ceiling is lowered (can only happen when a method of a message returns). Since messages are transferred from the timer-queue to the active-queue, a message with a higher priority than the currently executing message may be eligible for execution. If the head of the active-queue, $M_i$, has the highest priority, it is only allowed to execute if $\pi(M_i) > \Pi$ (see Section III). Assuming $\pi(M_i) > \Pi$, $M_i$ is transferred from the active-queue to the running-stack (containing all messages that have been allowed to start execution) and executed. If we instead assume $\pi(M_i) \leq \Pi$ then $M_i$ can only become eligible for execution when the system ceiling is lowered, i.e. when a message completes execution and releases the object-resource, thus a new scheduling decision must be made when execution of a message completes. Similarly, if $M_i$ does not have the highest priority it can only become eligible for execution when an asynchronous message completes execution. Synchronous messages that are generated by an executing message are always scheduled immediately, since SRP guarantees that all resources (objects) are available and the priority of a synchronous message is always inherited from the sending message.

C. Implementation of data structures

The data structures required by the kernel:

timer-queue: messages with a baseline in the future, sorted by ascending baseline
active-queue: messages with an expired baseline, sorted by descending priority
running-stack: messages that have been allowed to start execution
system ceiling: the maximum of object ceilings of locked objects

The active- and timer-queues are currently implemented as sorted lists. While there are more suitable data structures (e.g., heaps or balanced search trees), sorted lists are easier to implement and when the number of items in the list is small the performance is comparable to more advanced structures [20]. The running-stack is simply a last-in-first-out stack, and the system ceiling is implemented as an unsigned integer. To allow for efficient usage of common interrupt hardware (priority-based), some values of the system ceiling are implemented in hardware.

Example scheduling using interrupt hardware

A subset of the values of the system ceiling are implemented in hardware, i.e. the interrupt mask register of the processor. To demonstrate the benefits, we consider a simple example: let

$$\mathcal{M} = [M_1 = \text{ASYNC}(O_{\text{serial}}, m_{\text{read}}, B_{\text{inherited}}, 10\text{ms}),
M_2 = \text{ASYNC}(O_{\text{serial}}, m_{\text{write}}, B_{\text{inherited}}, 50\text{ms})]$$

From the definitions in Section III follows: $p(M_1) > p(M_2) \implies \pi(M_1) > \pi(M_2)$, and $[O_{\text{serial}}] = \pi(M_1)$. $M_1$ is sent from the data-ready interrupt of the serial port (with interrupt priority $p(M_1)$), and $M_2$ is sent from the ready-to-send interrupt (with interrupt priority $p(M_2)$). Since $B_{\text{inherited}}$ in the interrupt context corresponds to the time when the interrupt handler is invoked, the messages can be placed directly into the active-queue.

Let all possible values of the system ceiling that can be represented in hardware be defined as

$$\mathcal{H}, \mathcal{H} \subset \mathbb{N}_0$$

and the hardware system ceiling as

$$\Pi, \Pi \in \mathcal{H}$$

Then, assuming $\{\pi(M_1), \pi(M_2)\} \subset \mathcal{H}$, the messages can be placed directly into the running-stack. This follows from the one-to-one mapping between priority and preemption level, i.e. $p(M) = \pi(M)$, and for any messages $M_j$ and $M_k$ where $\{\pi(M_j), \pi(M_k)\} \subset \mathcal{H}$. If $M_j$ is currently executing and an interrupt handler (with priority $p(M_k)$) that generates $M_k$ is invoked, then $\Pi < \pi(M_k)$ (otherwise the interrupt would be masked by the hardware system ceiling) and $p(M_k)$ must be the highest priority since if $M_j$ is executing then $\Pi \geq \pi(M_j)$, and hence $p(M_k) > p(M_j)$. Thus, an interrupt handler is only invoked (and corresponding message generated) if it has the highest priority and all resources are available.

Discussion and limitations of implementation

In the previous example, we demonstrate how interrupt hardware is exploited to perform scheduling of messages. This hardware scheduling is limited by the number of interrupt priorities of the hardware. Thus, if no one-to-one mapping exists between message priorities and interrupt priorities, then co-operative software and hardware scheduling is required, i.e. two or more messages with different priorities must share a single interrupt priority. Whenever an interrupt priority is shared by different message priorities, the software scheduler
is invoked to determine if a generated message should be executed.

In the current implementation, transfer of time-delayed messages from timer-queue to active-queue is initiated by a timer interrupt. The priority of this interrupt must be set to that of the highest priority message in the timer-queue. Let \( p_i \) be the lowest priority of messages in the timer-queue, and \( p_h \) be the highest, then all messages (in the system) with priority \( p_m \), \( p_i < p_m < p_h \) will be subject to scheduling overhead (i.e. transfer of message from timer-queue to active-queue). However, if a system contains \( n \) time-delayed messages with \( p \) unique priorities (\( n \geq p \)), then this scheduling overhead can be mitigated by using \( p \) different timers, i.e. by minimizing the set of priorities in each timer-queue (one for each timer).

**VIII. CONCLUSION**

In this paper we have given an informal mapping from the undertaken CRC model to the notions of SRP. And shown how a CRC model can be translated into a CRO model. We have shown for an example system (process controller) how state, functionality and temporal properties are captured and abstracted in terms of the CRC model. We have proposed a method for code synthesis of CRC models, and shown how the timing requirements from the specification was preserved. We have proposed an algorithm that from a CRC model extracts resource ceilings and interrupt priorities for SRP-DM based scheduling. Additionally, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. The algorithm was exemplified on the process controller, showing how timing requirements from the model specification is translated into resource ceilings and interrupt priorities for the scheduler. Furthermore, we have presented the design of an SRP-DM kernel supporting cooperative hardware- and software-scheduling utilizing the derived resource ceilings and interrupt priorities at run-time.

**A. Current and future work**

Current and future work includes SRP based scheduling analysis for the presented model[21], with the aim to make safe schedulability estimations by taking the kernel overhead into consideration. Additionally we are investigating methods to derive and minimize the total memory requirement of a CRO system.

**ACKNOWLEDGMENT**

This work was supported in part by the Knowledge Foundation in Sweden under a research grant for the SAVE-IT project, the EU Interreg IV A North Programme (grant no. 304-15591-08), the ESI project (European Regional Development Fund, grant no. 41732), and the EU AESOP project (grant no. 258682).

**REFERENCES**


