Operational Semantics of Ada Ravenscar

Irfan Hamid and Elie Najm

Telecom ParisTech – LTCI-UMR 5141 CNRS
46, rue Barrault, 75013 – Paris, France
Irfan.Hamid@enst.fr, Elie.Najm@enst.fr

Abstract. The Ada programming language has been designed from the ground up for safety-critical real-time systems. This trend has continued with the Ada 2005 language definition which has incorporated the Ravenscar Profile for high-integrity systems into the language standard. Here we describe the operational semantics for Ada Ravenscar code generated automatically from an architecture description of the system given in the Architecture Analysis and Design Language.

1 Introduction

The Ada Ravenscar Profile [2] is a restriction of the rich tasking subset of the Ada language and associated runtime that aims to make the language more amenable to the development of safety-critical real-time systems. The Architecture Analysis and Design Language (AADL) [13] is an architecture description language targeted specifically to the real-time and avionics domain. The code generation rules given with the AADL standard are incomplete and rely on the existance of an "AADL executive", in effect, an operating system that provides all the services needed by an AADL application.

Since such an operating system does not exist, we used ORK [6], a Ravenscar-compliant executive. We developed code generation rules for AADL to Ada that faithfully preserve semantics when run on the ORK platform. We also developed a code generator as an Eclipse plugin (ARC `http://aadl.enst.fr/arc/`) that transforms AADL models to Ada source. The code generation rules and toolset were introduced in [9]. In this paper, we present a static semantics for the Ravenscar code that we generate, which is a subset of Ada Ravenscar. We also present a structured operational semantics that represents the dynamic evolution of the generated system. The Ravenscar Profile restrictions on Ada eliminate certain features from the language and associated runtime:

- All tasks must be either periodic or sporadic (for schedulability analysis)
- Tasks may only communicate among themselves through protected objects
- No dynamic creation or destruction of tasks or protected objects
- Rendezvous are prohibited (no entries on Ada tasks)
- Protected objects may have at most one entry
- A protected object entry's queue is of size 1
- All delays must be absolute (no `delay <time_expression>` allowed)

F. Kordon and T. Vardanega (Eds.): Ada-Europe 2008, LNCS 5026, pp. 44–58, 2008.

- Scheduling is priority based, priority assignment is RMA [15] or RTA [3]
- The priority ceiling protocol [16] is used for access to protected objects

ARC relies upon the OSATE AADL toolkit [14] to parse AADL models. The OSATE toolkit uses the Eclipse Modeling Framework [1] to represent the abstract syntax of the parsed model. Instead of directly generating Ada code from the AADL model, we chose to implement an intermediate meta-model to represent the Ravenscar system. The front-end transforms the AADL model to an instance of this meta-model. The code generator traverses this intermediate model—which we call the Ravenscar Meta-model (RMM)—to generate Ada code. Two advantages of this approach are a reduction in complexity (RMM is simpler than the AADL meta-model), and ease of writing code generators for other languages. Because all AADL models cannot be transformed to Ravenscar-compliant code, we verify the AADL model against a set of Object Constraint Language rules before a model transformation from AADL to an instance of the RMM is carried out. The paper is structured as follows. Sec. 2 presents the static semantics. Sec. 3 presents the dynamic semantics of the generated Ravenscar code using a structured operational semantic approach [12]. Sec. 4 relates our contribution to past and ongoing research and concludes.

2 Static Semantics

The static semantics provided in this section are a formalization of the structure of the RMM using set theory, and mirrors the static structure of code generated by ARC. This static semantics will be used in the ensuing section on dynamic semantics, specifically to manipulate the entities in the operational semantic transitions.

2.1 Ravenscar Computational Units

A Ravenscar system is given by five *finite* and *pairwise disjoint* sets, endowed with five functions and related by four relations. The five sets are:

$$\begin{aligned}
\textbf{Periodic tasks } \mathcal{T}_p &= \{P_1 \dots P_n\} \\
\textbf{Sporadic tasks } \mathcal{T}_s &= \{S_1 \dots S_m\} \\
\textbf{Interrupts } \mathcal{U} &= \{U_1 \dots U_k\} \\
\textbf{Synchronisers } \mathcal{D} &= \{D_1 \dots D_l\} \\
\textbf{Exchangers } \mathcal{E} &= \{E_1 \dots E_r\}
\end{aligned}$$

- *Sporadic tasks* are dispatched upon the reception of an event. A minimum time—characteristic to each task—between successive dispatches is enforced
- *Periodic tasks* are dispatched at regular time intervals called their *period*
- *Interrupts* can be raised at any time except if a previous occurence is already being executed. Thus, at any time, there can be at most $k = |\mathcal{U}|$ interrupts present in the system

- *Exchangers* are protected objects with an internal data buffer and `Get` and `Set` procedures. They are used for simple data exchange among tasks
- *Synchronisers* are protected objects with an internal queue of events that expose a `Send_Event` procedure for depositing events. A `Get_Event` *entry* is exposed upon which the associated sporadic task waits for dispatch

We define four derived sets, namely, **Tasks** ($\mathcal{T}$), **Activities** ($\mathcal{A}$), **Protected objects** ($\mathcal{PO}$), and **Computational units** ($\mathcal{C}$); as follows:

$$\begin{aligned} \mathcal{T} &= \mathcal{T}_p \cup \mathcal{T}_s \\ \mathcal{A} &= \mathcal{T}_p \cup \mathcal{T}_s \cup \mathcal{U} \\ \mathcal{PO} &= \mathcal{E} \cup \mathcal{D} \\ \mathcal{C} &= \mathcal{A} \cup \mathcal{PO} \end{aligned}$$

2.2 Functions on Computational Units

Five functions on computation units are defined with the following signatures:

$$\text{PRIORITY} : \mathcal{C} \rightarrow \text{ANYPRIORITY} \quad (1)$$

$$\text{HOLDINGTIME} : \mathcal{T} \rightarrow \text{TIME} \quad (2)$$

$$\text{PROG} : \mathcal{C} \rightarrow \text{PROGS} \quad (3)$$

TIME is a discrete time domain. HOLDINGTIME is defined as the period for a periodic and minimum inter-dispatch time for a sporadic task. ANYPRIORITY is a bounded subset of the set $\mathbb{N}$ of natural numbers and gives valid priorities. PROGS is the subset of Ada 95 that the code of computational units conforms to. The code of a computational unit γ is given by PROG(γ).

2.3 Conformant PROGS Programs

We focus on an abstraction of programs that represents execution steps relevant to our semantics, which gives legal instructions and their sequencing:

- `comp`: A sequential execution step
- `Set`(E): A Set call to exchanger E
- `Get`(E): A Get call to exchanger E
- `Send_Event`(D): A Send_Event call to synchroniser D
- `Get_Event`(D): A Get_Event call to synchroniser D
- `delay until`: request to be suspended until a future instant
- `ret`: `return` statement

The legal execution sequences of these steps depend on the type of the computational unit. They are defined using BNF grammars. The code of each computational unit must respect its prescribed grammar (BP is for periodic tasks, BS for sporadic, BU for interrupts, BE for exchangers, BD for synchronizers):

BP := comp; BP | Set(E); BP | Get(E); BP | Send_Event(D); BP | delay until
BS := Get_Event(D); BP
BU := Send_Event(D) | Set(E)
BE := [Set $->$ CC , Get $->$ CC]
BD := [Send_Event $->$ CC, Get_Event $->$ CC]
CC := comp ;CC | ret

2.4 Topological Relations on Computational Units

By an analysis of the set of programs PROG, we can construct the communication topology between the various computational units. Four topological relations, *sets*, *gets*, *sends_event*, and *gets_event* are induced by PROG:

$$sets : \xrightarrow[\texttt{Set}]{} \subset \mathcal{A} \times \mathcal{E} \tag{4}$$

$$gets : \xrightarrow[\texttt{Get}]{} \subset \mathcal{T} \times \mathcal{E} \tag{5}$$

$$sends\_event : \xrightarrow[\texttt{Send\_Event}]{} \subset \mathcal{A} \times \mathcal{D} \tag{6}$$

$$gets\_event : \xrightarrow[\texttt{Get\_Event}]{} \subset \mathcal{T}_s \times \mathcal{D} \tag{7}$$

they are defined according to the following conditions:

$$\texttt{Set}(E) \text{ occurs-in } \text{PROG}(\alpha) \quad \Leftrightarrow \quad \alpha \xrightarrow[\texttt{Set}]{} E \tag{8}$$

$$\texttt{Get}(E) \text{ occurs-in } \text{PROG}(T) \quad \Leftrightarrow \quad T \xrightarrow[\texttt{Get}]{} E \tag{9}$$

$$\texttt{Send\_Event}(D) \text{ occurs-in } \text{PROG}(\alpha) \quad \Leftrightarrow \quad \alpha \xrightarrow[\texttt{Send\_Event}]{} D \tag{10}$$

$$\texttt{Get\_Event}(D) \text{ occurs-in } \text{PROG}(S) \quad \Leftrightarrow \quad S \xrightarrow[\texttt{Get\_Event}]{} D \tag{11}$$

We also need three derived relations, namely: *dispatches* ($\xrightarrow[DIS]{}$), *writes_to* ($\xrightarrow[WTO]{}$) and *accesses* ($\xrightarrow[ACC]{}$). *dispatches* is the inverse of *gets_event*, *writes_to* is the union of *gets* and *sends_event*, and *accesses* is the union of the four primitive relations. Formally:

$$D \xrightarrow[DIS]{} S \triangleq S \xrightarrow[\texttt{Get\_Event}]{} D \tag{12}$$

$$\alpha \xrightarrow[WTO]{} \pi \triangleq (\pi \in \mathcal{E} \ \wedge \ \alpha \xrightarrow[\texttt{Set}]{} \pi) \vee (\pi \in \mathcal{D} \ \wedge \ \alpha \xrightarrow[\texttt{Send\_Event}]{} \pi) \tag{13}$$

$$\begin{aligned} \alpha \xrightarrow[ACC]{} \pi \triangleq\ & (\pi \in \mathcal{E} \ \wedge \ \alpha \xrightarrow[\texttt{Set}]{} \pi) \vee (\pi \in \mathcal{E} \ \wedge \ \alpha \xrightarrow[\texttt{Get}]{} \pi) \\ & \vee \ (\alpha \in \mathcal{A} \ \wedge \ \pi \in \mathcal{D} \ \wedge \ \alpha \xrightarrow[\texttt{Send\_Event}]{} \pi) \\ & \vee \ (\alpha \in \mathcal{T}_s \ \wedge \ \pi \in \mathcal{D} \ \wedge \ \alpha \xrightarrow[\texttt{Get\_Event}]{} \pi) \end{aligned} \tag{14}$$

The topological relations must satisfy the following constraints:

$$\forall\ D,\ \exists\ S \text{ unique satisfying: } D \overline{\ _{DIS}\ } S \tag{15}$$

$$\forall\ S,\ \exists\ D \text{ unique satisfying: } S \overline{\ _{\texttt{Get\_Event}}\ } D \tag{16}$$

$$\forall\ U,\ \exists\ \pi \text{ unique satisfying: } U \overline{\ _{WTO}\ } \pi \tag{17}$$

$$U \overline{\ _{WTO}\ } \pi \ \text{ and } \ U' \overline{\ _{WTO}\ } \pi \ \Rightarrow \ U = U' \tag{18}$$

At most one task is dispatched by a synchronizer (15). For every sporadic task, there exists one and only one synchronizer that dispatches it (16). Each interrupt writes on one and only one protected object (17). At most one interrupt may write to a protected object (18). Constraints (15) and (16) imply that relations $\overline{\ _{DIS}\ }$ and $\overline{\ _{\texttt{Get\_Event}}\ }$ are bijective and mutually inverse functions. From (17) and (18) it follows that relation $\overline{\ _{WTO}\ }$, when restricted to $\mathcal{U}$, is an injective function with co-domain in $\mathcal{PO}$.

Priority Ceiling Protocol. All priorities must comply with PCP. Function PRIORITY must satisfy the following property ($\overline{\ _{ACC}\ }$ from equation 14):

$$\text{For any activity } \alpha \text{ and any protected object } \pi : \\ (\alpha \overline{\ _{ACC}\ } \pi) \ \Rightarrow \ \text{PRIORITY}(\pi) \geq \text{PRIORITY}(\alpha) \tag{19}$$

3 Dynamic Semantics

The dynamic semantics of the system will be described using a form of *structured operational semantics* [12] which describes the evolution of the system over time.

3.1 Execution Context

$$\textit{Execution context } \ c = \begin{cases} \sigma, \sigma_s & \textit{Scheduler} \\ \iota & \textit{Idle task} \\ a & \textit{An active execution context} \end{cases}$$

The above equation states that three entities may possess processing resources, the scheduler (σ and σ_s), the system idle task (ι), or an activity (a). The scheduler can be in one of two states: σ when the scheduler has seized control, σ_s when the scheduler is ready to grant control. Thus, σ and σ_s represent two steps in the excuction of the scheduler functions, allowing the assignment of different execution times to both in order to accurately model context switches. An active context, a, may have one of the following forms:

$$\begin{array}{ll} \textit{Sporadic tasks:} & S,\ S \overline{\ _{\texttt{Set}}\ } E,\ S \overline{\ _{\texttt{Send\_Event}}\ } D,\ S \overline{\ _{\texttt{Get}}\ } E,\ S \overline{\ _{\texttt{Get\_Event}}\ } D \\ \textit{Periodic tasks:} & P,\ P \overline{\ _{\texttt{Set}}\ } E,\ P \overline{\ _{\texttt{Send\_Event}}\ } D,\ P \overline{\ _{\texttt{Get}}\ } E \\ \textit{Interrupts:} & U,\ U \overline{\ _{\texttt{Set}}\ } E,\ U \overline{\ _{\texttt{Send\_Event}}\ } D \end{array}$$

The PRIORITY function is extended to active contexts and in conformance with the priority ceiling protocol, as follows (where the form $\alpha \underset{x}{\multimap} \pi$ corresponds to the context of a protected object π executing a call x issued by activity α):

$$\text{PRIORITY}(\alpha \underset{x}{\multimap} \pi) = \text{PRIORITY}(\pi) \tag{20}$$

We use a record notation—as defined in [4]—to maintain the state information of computational units. The fields corresponding to each computational unit are given in Table 1. We will use the dot notation to extract fields from records. $T{\cdot}\mathsf{Beh}$ is the value of field Beh in the record T. The update of a field in a record is performed as in the following example where D' is the record obtained by updating in record D the field Bar with true and field Queue with ϵ:

$$D' = \langle D \leftarrow \mathsf{Bar} = \mathsf{true} \leftarrow \mathsf{Queue} = \epsilon \rangle$$

Table 1. Fields present in state records of Ravenscar Computational Units

Description of field	Name	Type	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{T}_s$	$\mathcal{T}_p$	$\mathcal{U}$
Current program state	Beh	PROGS	√	√	√	√	√
Next dispatching time	Nd	TIME			√	√	
Elapsed time	Et	TIME			√	√	√
Processing time	Pt	TIME			√	√	√
Queue on entry	Queue	$\mathcal{T}_s \cup \{\epsilon\}$	√				
Barrier state	Bar	BOOL	√				
Event count	Ec	$\mathbb{N}$	√				

3.2 Ready Queue

A ready queue, R, is made of a (possibly empty) sequence of active execution contexts. We use $\circ$ as a sequence operator, hence, if a is an execution context and R a ready queue then $(a \circ R)$ is a ready queue whose head is a and whose tail is R. The empty ready queue will be denoted by ϵ. Ready queues satisfy the *priority-ordered* property, which is inductively defined as follows:

(i) ϵ is *priority-ordered*

(ii) $a \circ R$ is *priority-ordered* iff:
- R is *priority-ordered* and
- $\forall a' \in R :\ \text{PRIORITY}(a') \leq \text{PRIORITY}(a)$

The satisfaction by a queue R of the *priority-ordered* property implies that R is an ordered list of queues having the form: $R = r_{p_1} \circ \ldots \circ r_{p_n}$ where for each r_{p_n}:

$$\forall i, j : i < j \ \Rightarrow\ p_i > p_j \qquad \text{and} \tag{21}$$

$$\forall i,\ \forall a \in r_{p_i} : \text{PRIORITY}(a) = p_i \tag{22}$$

All active contexts in the same subqueue have the same priority (22), and subqueues are ordered according to their priorities (21). We define priority head insertion and priority tail insertion for ready queues as both methods are used:

Let $p_k = \text{PRIORITY}(a)$
Priority Head Insertion $a_{\odot}R \;=$

$$\begin{array}{l} r_{p_1} \circ \ldots \circ a \circ r_{p_k} \circ \ldots \circ r_{p_n} \text{ when } R = r_{p_1} \circ \ldots \circ r_{p_k} \circ \ldots \circ r_{p_n} \\ r_{p_1} \circ \ldots \circ r_{p_i} \circ a \circ r_{p_j} \;\ldots\; r_{p_n} \text{ when } R = r_{p_1} \circ \ldots \circ r_{p_i} \circ r_{p_j} \circ \ldots \circ r_{p_n} \wedge \; p_i < p_k < p_j \end{array} \tag{23}$$

Priority Tail Insertion $R_{\odot}a \;=$

$$\begin{array}{l} r_{p_1} \circ \ldots \circ r_{p_k} \circ a \circ \ldots \circ r_{p_n} \;\text{ when }\; R = r_{p_1} \circ \ldots \circ r_{p_k} \circ \ldots \circ r_{p_n} \\ r_{p_1} \circ \ldots \circ r_{p_i} \circ a \circ r_{p_j} \;\ldots\; r_{p_n} \text{ when } R = r_{p_1} \circ \ldots \circ r_{p_i} \circ r_{p_j} \circ \ldots \circ r_{p_n} \wedge \; p_i < p_k < p_j \end{array} \tag{24}$$

A task taken from the blocked set to the ready queue is inserted at the tail of the ready queue for its priority, whereas one that is preempted during execution by the scheduler is inserted at the head of the ready queue for its priority.

3.3 Structure of the State of a Ravenscar System

The state of a Ravenscar system has a static part made up of the set of records of all computational units, and a dynamic part which is given by the vector:

$$IL \rightsquigarrow \left[c, R, B, \mathsf{ns}, \mathsf{t}\right] \tag{25}$$

- IL: list of interrupts present in the system, waiting to be handled. When the list of interrupts is empty, the leading "$IL \rightsquigarrow$" may be ommitted
- c: current execution context
- R: ready queue
- B: set of blocked tasks
- ns: time of the next system clock tick when control is passed to the scheduler
- t: current time, i.e.: the current age of the system

Each of the execution context types (scheduler, idle, or active) may perform specific execution steps. These steps cause the state of the system to evolve over time. The steps performed by the active context depend on the current state of the code of its activity, given by the Beh field of the state record of the activity. The steps performable by the scheduler are: (i) *suspending activity a and taking control* $(a \xrightarrow{\text{as}} \sigma)$; (ii) *suspending idle task and taking control* ($(\iota \xrightarrow{is} \sigma)$; (iii) *self suspention to handle interrupts* $(\sigma_s \xrightarrow{\text{ss}} \sigma)$; (iv) *handling an interrupt* $(\sigma \xrightarrow{\text{ih}} \sigma)$; (v) *updating the ready queue* $(\sigma \xrightarrow{\text{ud}} \sigma_s)$; (vi) *granting control to activity a* $(\sigma_s \xrightarrow{\text{sa}} a)$; (vii) *granting control to idle task* $(\sigma_s \xrightarrow{\text{si}} \iota)$. The idle task performs one type of steps which is *idling*: $(\sigma_s \xrightarrow{\text{idling}} \iota)$.

3.4 Initial State of a Ravenscar System

The initial state of a Ravenscar system is given by:

$$\left[\sigma, R_0, B_0, 0, 0\right] \tag{26}$$

where the initial ready queue, R_0, is a *priority-ordered* list of all tasks: $R_0 = T_1 \circ \ldots \circ T_n$, and B_0 the initial set of blocked tasks is an empty set: $B_0 = \{\}$. Moreover, the initial state of each of the periodic tasks, the sporadic tasks, the synchronisers and the exchangers, is given by their associated records:

$$P = \langle\ \mathsf{Beh} = \textsc{prog}(P), \mathsf{Nd} = 0, \mathsf{Et} = 0, \mathsf{Pt} = 0\ \rangle$$
$$S = \langle\ \mathsf{Beh} = \textsc{prog}(S), \mathsf{Nd} = 0, \mathsf{Et} = 0, \mathsf{Pt} = 0\ \rangle$$
$$E = \langle\ \mathsf{Beh} = \textsc{prog}(E)\ \rangle$$
$$D = \langle\ \mathsf{Beh} = \textsc{prog}(D), \mathsf{Queue} = \epsilon, \mathsf{Ec} = 0, \mathsf{Bar} = \mathsf{false}\ \rangle$$

3.5 State Transitions of a Ravenscar System

The execution of a Ravenscar system is given by the set of structured operational semantics rules, having the structure of a fraction:

$$\frac{\textit{Antecedents}}{IL \rightsquigarrow \big[c, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{act} IL' \rightsquigarrow \big[c', R,' B', \mathsf{ns}', \mathsf{t}\big] \ \hat{+}\ \delta(act)} \quad \textit{SHORT NAME}$$

Antecedents (numerator) are conditions which need to hold for the *Consequent* (denominator) part to be applied. *Antecedents* depend on the current state of the system. *Consequent* part denotes the transition taken and the action—*act*—performed. *act* represents the smallest possible uninterruptible instruction. It is an indivisible unit; interrupts will either be fired before or after such an instruction. Complex instructions like `delay until` are considered a sequence of simpler instructions with the final indivisible one actually having the intended impact. *IL* and *IL'* are optional, they represent the list of interrupts present before and after the transition. $\delta(act)$ is the time consumed by the transition, and $\hat{+}\ \delta(act)$ is the *ageing* operator. It is formally defined as follows:

$$\big[c, R, B, \mathsf{ns}, \mathsf{t}\big] \ \hat{+}\ \delta = \big[c \ \hat{+}\ \delta,\ R \ \hat{+}\ \delta,\ B,\ \mathsf{ns},\ \mathsf{t} + \delta\big]$$

where:

$$c \ \hat{+}\ \delta = \begin{cases} \sigma & \text{if} \quad c = \sigma \\ \iota & \text{if} \quad c = \iota \\ \langle \alpha \leftarrow \mathsf{Et} = \alpha \cdot \mathsf{Et} + \delta \leftarrow \mathsf{Pt} = \alpha \cdot \mathsf{Pt} + \delta \rangle & \text{if} \quad c = \alpha \vee c = \alpha \underline{\ }_x \pi \end{cases}$$

$$R \ \hat{+}\ \delta = \langle a_1 \leftarrow \mathsf{Et} = a_1.\mathsf{Et} + \delta \rangle \circ \ldots \circ \langle a_n \leftarrow \mathsf{Et} = a_n.\mathsf{Et} + \delta \rangle \text{ for } R = a_1 \circ \ldots \circ a_n$$

The above equations state that if the currently executing task is either the scheduler or the idle task then the ageing operator has no effect on it. However, if the excution context is an active one then the ageing operator adds the $\delta(action)$ amount of time to both the elapsed time (Et) and processing time (Pt) fields of the record of the activity. On the other hand, for all tasks in the ready queue R, the ageing operator only adds the $\delta(action)$ amount of time to the elapsed time field (they are not budgeted for this time). We now provide the transition rules, starting with the system idle task and ending with rules for interrupt handling.

Idling: Rule *IDLE* shows the idle task executing. The antecedent shows that the system can only idle if it hasn't reached the next scheduling instant ns. The *age* of the system advances by an amount $\delta(\text{idling})$.

$$\frac{\mathsf{t} < \mathsf{ns}}{\big[\iota, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{idling}} \big[\iota, R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{idling})} \; \textit{IDLE}$$

Pure Computation Steps: The *CMPT* and *CMPO* transitions represent sequential computations that have no side-effects on tasking or inter-task communication. *CMPT* denotes a task carrying out a sequential computation, *CMPO* denotes a protected object carrying out a sequential computation. The behavior (Beh) must in both cases have `comp` instruction at the head, the current time must be less than the next dispatching time for the scheduler.

$$\frac{T \cdot \mathsf{Beh} = \mathsf{comp}; \mathsf{C} \;\wedge\; \mathsf{t} < \mathsf{ns}}{\big[T, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{comp}} \big[T', R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\mathsf{comp})} \; \textit{CMPT}$$
$$T' = \langle T \leftarrow \mathsf{Beh} = \mathsf{C} \rangle$$

$$\frac{\pi \cdot \mathsf{Beh} = \mathsf{comp}; \mathsf{C} \;\wedge\; \mathsf{t} < \mathsf{ns}}{\big[\alpha \underline{\ }_x \pi, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{comp}} \big[\alpha \underline{\ }_x \pi', R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\mathsf{comp})} \; \textit{CMPO}$$
$$\pi' = \langle \pi \leftarrow \mathsf{Beh} = \mathsf{C} \rangle$$

Protected Objects: The rule *NBCL* represents an activity (task or interrupt) calling a procedure of a protected object. The antecedent states that the current behaviour of the activity is a call to a procedure, and that the current time is less than the next scheduler launching time. The consequent is that the code of the protected object is being executed in the context of the activity α ($\alpha' \underline{\ }_x \pi$).

$$\frac{\begin{array}{c}\alpha \cdot \mathsf{Beh} = x(\pi); C \;\wedge \\ x \in \{\text{Get, Set, Send\_Event}\} \;\wedge\; \mathsf{t} < \mathsf{ns}\end{array}}{\big[\alpha, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{x} \big[\alpha' \underline{\ }_x \pi, R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(x)} \; \textit{NBCL}$$
$$\alpha' = \langle \alpha \leftarrow \mathsf{Beh} = C \rangle$$
$$\pi' = \langle \pi \leftarrow \mathsf{Beh} = \text{PROG}(\pi).x \rangle$$

The transitions *RET1* through *RET4* depict how calls from protected objects return. *RET1* represents the `return` from a protected object procedure. The consequent shows that the execution time is budgeted to the task's processing time Pt. The calling activity is placed at the head of the ready queue and the scheduler takes control to evaluate barriers. *RET2* shows a synchronizer returning from a `Send_Event` procedure when the entry queue is empty. Transition *RET3* shows a synchronizer returning from a `Send_Event` procedure when

the entry queue is *not* empty. The blocked `Get_Event` entry is immediately executed in the context of the task waiting on it. *RET4* gives the situation where a synchronizer returns from `Get_Event` entry call. The task in whose context the execution was taking place is preempted and is placed at the head of its ready queue, and the scheduler takes over.

$$\frac{E \cdot \mathsf{Beh} = \mathsf{ret} \ \wedge \ x \in \{\mathrm{Get}, \mathrm{Set}\} \ \wedge \ \mathsf{t} < \mathsf{ns}}{\left[\alpha \underset{x}{\frown} E, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\mathsf{ret}} \left[\sigma, \alpha' \odot R, B, \mathsf{ns}, \mathsf{t}\right] \hat{+} \delta(\mathsf{ret})} \quad \textit{RET1}$$

$$\alpha' = \langle \alpha \leftarrow \mathsf{Pt} = \alpha \cdot \mathsf{Pt} + \delta(\mathsf{ret}) \rangle$$

$$\frac{D \cdot \mathsf{Beh} = \mathsf{ret} \ \wedge \ D \cdot \mathsf{Queue} = \epsilon \ \wedge \ \mathsf{t} < \mathsf{ns}}{\left[\alpha \underset{\texttt{Send\_Event}}{\frown} D, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\mathsf{ret}} \left[\sigma, \alpha' \odot R, B, \mathsf{ns}, \mathsf{t}\right] \hat{+} \delta(\mathsf{ret})} \quad \textit{RET2}$$

$$D' = \langle D \leftarrow \mathsf{Bar} = \mathrm{true} \leftarrow \mathsf{Ec} = D \cdot \mathsf{Ec} + 1 \rangle$$
$$\alpha' = \langle \alpha \leftarrow \mathsf{Pt} = \alpha' \cdot \mathsf{Pt} + \delta(\mathsf{ret}) \rangle$$

$$\frac{D \cdot \mathsf{Beh} = \mathsf{ret} \ \wedge \ D \cdot \mathsf{Queue} = S \ \wedge \ \mathsf{t} < \mathsf{ns}}{\left[\alpha \underset{\texttt{Send\_Event}}{\frown} D, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\mathsf{ret}} \left[S' \underset{\texttt{Get\_Event}}{\frown} D', \alpha' \odot R, B', \mathsf{ns}, \mathsf{t}\right] \hat{+} \delta(\mathrm{ret})} \quad \textit{RET3}$$

$$B' = B \setminus \{S\}$$
$$S' = \langle S \leftarrow \mathsf{Nd} = \mathsf{t} + \textsc{holdingtime}(S) \rangle$$
$$D' = \langle D \leftarrow \mathsf{Bar} = \mathrm{true} \leftarrow \mathsf{Ec} = D \cdot \mathsf{Ec} + 1 \rangle$$
$$\alpha' = \langle \alpha \leftarrow \mathsf{Pt} = \alpha \cdot \mathsf{Pt} + \delta(\mathsf{ret}) \rangle$$

$$\frac{D \cdot \mathsf{Beh} = \texttt{ret; C} \ \wedge \ \mathsf{t} < \mathsf{ns}}{\left[S \underset{\texttt{Get\_Event}}{\frown} D, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\mathsf{ret}} \left[\sigma, S' \odot R, B, \mathsf{ns}, \mathsf{t}\right] \hat{+} \delta(\mathsf{ret})} \quad \textit{RET4}$$

$$D' = \langle D \leftarrow \mathsf{Bar} = (D \cdot \mathsf{Ec} > 1) \leftarrow \mathsf{Ec} = D \cdot \mathsf{Ec} - 1 \leftarrow \mathsf{Queue} = \epsilon \rangle$$
$$S' = \langle S \leftarrow \mathsf{Pt} = S \cdot \mathsf{Pt} + \delta(\mathsf{ret}) \rangle$$

Rules *OBCL* and *CBCL* represent a sporadic task issuing a `Get_Event` call. Rule *OBCL* represents when the barrier is open and the call is immediately executed. Rule *CBCL* represents when the barrier is closed, the call remains blocked on the entry until a `Set_Event` is issued by another task or interrupt.

$$\frac{S \cdot \mathsf{Beh} = \texttt{Get\_Event(}D\texttt{); C} \ \wedge \ D \cdot \mathsf{Bar} = \mathrm{True} \ \wedge \ \mathsf{t} < \mathsf{ns}}{\left[S, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\mathrm{Get\_Event}} \left[S' \underset{\texttt{Get\_Event}}{\frown} D', R, B, \mathsf{ns}, \mathsf{t}\right] \hat{+} \delta(\mathrm{Get\_Event})} \quad \textit{OBCL}$$

$$S' = \langle S \leftarrow \mathsf{Beh} = C \leftarrow \mathsf{Nd} = \mathsf{t} + \textsc{holdingtime}(S) \rangle$$
$$D' = \langle D \leftarrow \mathsf{Beh} = \textsc{prog}(D) \cdot \texttt{Get\_Event} \rangle$$

$$\frac{S{\cdot}\mathsf{Beh} = \texttt{Get\_Event(}D\texttt{);}\ \texttt{C} \wedge D{\cdot}\mathsf{Bar} = \text{False} \wedge\ \mathsf{t} < \mathsf{ns}}{\big[S, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{Get\_Event}} \big[S', R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{Get\_Event})} \quad \textit{CBCL}$$

$S' = \langle S \leftarrow \mathsf{Beh} = C \leftarrow \mathsf{Bar} = \text{true}\rangle$
$D' = \langle D \leftarrow \mathsf{Queue} = S\rangle$

Scheduler. The scheduler also takes control at certain points called *scheduling points*. Some of these have already been explained (the RET_i transitions). Others occur when the active context executes a `delay until` instruction, and when the scheduler is *scheduled* to execute, represented by the `ns` variable in the system configuration and calculated just before the scheduler cedes control. *NS-IDLE* and *NS-ACT* represent the scheduler preempting the idle task and an activity (respectively) as its launch time arrives. *SDELAY* and *PDELAY* show a sporadic task and a periodic task (respectively) execute a `delay until`. `ns` is the minimum of `Nd` fields of all tasks in the blocked set where `Nd` represents the next dispatching time for the task: $\mathtt{ns} = \min_{T_i \in B}(T_i \cdot \mathtt{Nd})$. *SCUD* is the evolution of the scheduler as it evaluates and updates the ready queue and blocked tasks. *SCAC* shows the scheduler calculating its next dispatching time and then granting control to the highest priority ready task. *SCID* is the action carried out by the scheduler when the ready queue is empty.

$$\frac{\mathsf{ns} \leq \mathsf{t}}{\big[\iota, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{is}} \big[\sigma, R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{is})} \quad \textit{NS-IDLE}$$

$$\frac{\mathsf{ns} \leq \mathsf{t}}{\big[a, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{as}} \big[\sigma, a_{\circ}R, B, \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{as})} \quad \textit{NS-ACT}$$

$$\frac{T{\cdot}\mathsf{Beh} = \texttt{delay;C} \wedge\ \mathsf{t} < \mathsf{ns}}{\big[T, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{delay}} \big[\sigma, R, B', \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{delay})} \quad \textit{SDELAY}$$

$T' = \langle T \leftarrow \mathsf{Beh} = \text{PROG}(T) \leftarrow \mathsf{Pt} = T \cdot \mathsf{Pt} + \delta(\text{delay})\rangle$
$B' = B \cup \{T'\}$

$$\frac{T{\cdot}\mathsf{Beh} = \texttt{delay ; C,}\quad \mathsf{t} < \mathsf{ns}}{\big[T, R, B, \mathsf{ns}, \mathsf{t}\big] \xrightarrow{\text{delay}} \big[\sigma, R, B', \mathsf{ns}, \mathsf{t}\big] \,\hat{+}\, \delta(\text{delay})} \quad \textit{PDELAY}$$

$T' = \langle T \leftarrow \mathsf{Beh} = \text{PROG}(T) \leftarrow \mathsf{Pt} = T \cdot \mathsf{Pt} + \delta(\text{delay})$
$\qquad \leftarrow Nd = T \cdot Nd + \text{HOLDINGTIME}(T)\rangle$
$B' = B \cup \{T'\}$

$$\frac{-}{\left[\sigma, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{ud}} \left[\sigma_s, R', B', \mathsf{ns}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{ud})} \; \textit{SCUD}$$

$$B' = B \setminus \text{ready}(B, \mathsf{t})$$
$$R' = R \odot \text{ready}(B, \mathsf{t})$$
$$\text{ready}(B, \mathsf{t}) = \{T \in B \;\mid\; T{\cdot}\mathsf{Nd} \geq t\}$$

$$\frac{-}{\left[\sigma_s, a \circ R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{sa}} \left[a, R, B, \mathsf{ns'}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{sa})} \; \textit{SCAC}$$

$$\mathsf{ns'} = \text{Min}_{T \in B}(T{\cdot}\mathsf{Nd})$$

$$\frac{-}{\left[\sigma_s, \epsilon, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{si}} \left[\iota, \epsilon, B, \mathsf{ns'}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{si})} \; \textit{SCID}$$

$$\mathsf{ns'} = \text{Min}_{T \in B}(T{\cdot}\mathsf{Nd})$$

Interrupt Handling. Rule *NEWI* models the arrival of a new interrupt. *I-AS* and *I-US* depict the scheduler preempting an activity and an idle task (respectively) in presence of interrupts in order to handle them. In case of arrival of interrupt during interrupt handling by the scheduler, the scheduler is restarted (transition *I-SS*). *I-IH* depicts the scheduler selecting the highest priority interrupt and inserting it at the tail of its priority list in the ready queue (all interrupt priorities are greater than all task priorities so an interrupt *will* preempt a task).

$$\frac{U \notin (\{c\} \cup R \cup IL)}{IL \rightsquigarrow \left[c, R, B, \mathsf{ns}, \mathsf{t}\right] \longrightarrow IL \circ U \rightsquigarrow \left[c, R, B, \mathsf{ns}, \mathsf{t}\right]} \; \textit{NEWI}$$

$$\frac{IL \neq \phi}{IL \rightsquigarrow \left[a, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{as}} IL \rightsquigarrow \left[\sigma, a \circ R, B, \mathsf{ns}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{as})} \; \textit{I-AS}$$

$$\frac{IL \neq \phi}{IL \rightsquigarrow \left[\iota, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{is}} IL \rightsquigarrow \left[\sigma, R, B, \mathsf{ns}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{is})} \; \textit{I-IS}$$

$$\frac{IL \neq \phi}{IL \rightsquigarrow \left[\sigma_s, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{ss}} IL \rightsquigarrow \left[\sigma, R, B, \mathsf{ns}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{ss})} \; \textit{I-SS}$$

$$\frac{-}{U \circ IL \rightsquigarrow \left[\sigma, R, B, \mathsf{ns}, \mathsf{t}\right] \xrightarrow{\text{ih}} IL \rightsquigarrow \left[\sigma, R \odot U, B, \mathsf{ns}, \mathsf{t}\right] \,\hat{+}\, \delta(\text{ih})} \; \textit{I-IH}$$

3.6 Discussion

One of the outcomes of providing a formal semantics (of a model, language or algorithm) is that it allows to disambiguate the informal description in the natural language. One can thus formally reason about properties of the system thus described. As an example, in our semantics, we made a choice in the way the inter-dispatch time is computed. If we had strictly obeyed the Ravenscar code patterns, we would have used a modified syntax for sporadic tasks in $\mathbb{PROGS}$ whereby we would have made explicit the capture of the current clock from the system. We would have also had to decompose rules *RET3* and *CBCL*, introducing an additional step reflecting the capture of the current clock. A small discrepancy would then arise due to the non-atomic nature of the sporadic task release and computation of the next dispatch time, i.e., a higher priority task may preempt the sporadic task between these two actions. In case of a preemption by a higher priority task between the release of the task and the computation of the next release, a longer than stipulated inter-dispatch time may be enforced. This does not impact schedulability but can result in the sporadic tasks responding more sluggishly. This problem can be solved by assigning synchronizers the maximum priority in the system (`Max_Interrupt_Priority`), and returning the instance of time when the entry is executed. The maximum priority ensures that a task *cannot* be preempted while it is in the entry, thus ensuring the atomicity of the two actions. In our semantics, we chose a solution whereby the computation of the next release of sporadic tasks is performed as a side effect of the sporadic task entering the synchroniser. One may think of the scheduler performing this computation. Indeed, although the scheduler is not explicitely stated in rules rules *RET3* and *CBCL*, nevertheless, it is the scheduler which is responsible for granting control of the sporadic task when it enters the synchroniser. Thus, the scheduler performs the computation in an atomic fashion.

4 Conclusions

Previously, work has been undertaken ([8], [10], [17] and [7]) to formalize the semantics of real-time kernels and Ravenscar-like executives. In [7], the author defines an extension of CCS aimed at studying muti-tasking systems. Similarly to our approach, the general behaviour of systems made of concurrent tasks can be modeled. However, in our work, we represent the kernel functions explicitly, allowing us to account for system overhead. The work that is the closest to ours is perhaps [8] where the authors use the RTL and PVS formalisms to develop a Ravenscar-like kernel. A major difference with our contribution is that [8] aims at prescribing the development of the kernel functions whereas our contribution does provide an operational semantics which captures the global behaviour of Ravenscar systems (composed from the kernel and the running application). Another main difference with existing work is that our paper is the first direct approach at providing semantics using the structured operational semantics and not requiring any other notational support. In [11], the authors present a timed automata-based approach to the verification of Ravenscar systems.

The structured operational semantic formalization of RMM is pivotal to our tool chain as it provides to developers a direct and unambiguous description of the running behaviour of their hard real-time applications. Our semantics helped also to explicitly define the kernel functions and scheduler overheads due to context switches and interrupt handling. While the AADL is an architecture description language with open and loose semantics, our AADL to RMM transformation tool determines a rigourous semantic definition for a subset of AADL (the subset that is translatable into RMM). It must be kept in mind that the semantics given here are for executable systems generated from AADL models that are to run on a Ravenscar executive.

As stated in the introduction, with the RMM semantics we have a complete and unambiguous description of the interaction of functional code with the generated framework. This is possible due to the abstraction of functional code into the set of PROGS legal programs for all units. The work achieved can be usefully extended according to the approach of "semantic anchoring", whereby our operational semantics could be transposed using Abstract State Machines as a supporting anchoring language [5].

Acknowledgements. Work funded via the ASSERT project (IST-FP6-2004 004033) funded in part by the European Commission. We would like to thank Tullio Vardanega and Juan Antonio de la Puente for their insightful remarks.

References

1. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2004)
2. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. Ada Lett. XXIV(2), 1–74 (2004)
3. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages, 3rd edn. Addison-Wesley, Reading (2001)
4. Cardelli, L., Mitchell, J.C.: Operations on Records. In: Proceedings of the fifth international conference on Mathematical Foundations of Programming Semantics, pp. 22–52. Springer, New York (1990)
5. Chen, K., Sztipanovits, J., Neema, S.: Toward a Semantic Anchoring Infrastructure for Domain-specific Modeling Languages. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 35–43. ACM Press, New York (2005)
6. de la Puente, J.A., Ruiz, J.F., Zamorano, J.: An Open Ravenscar Real-Time Kernel for GNAT. In: Keller, H.B., Plödereder, E. (eds.) Ada-Europe 2000. LNCS, vol. 1845, pp. 5–15. Springer, Heidelberg (2000)
7. Fidge, C.J.: The Algebra of Multi-tasking. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 213–227. Springer, Heidelberg (2000)
8. Fowler, S., Wellings, A.: Formal Development of a Real-Time Kernel. In: RTSS 1997: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997), p. 220. IEEE Computer Society, Washington, DC (1997)
9. Hamid, I., Zalila, B., Najm, E., Hugues, J.: A Generative Approach to Building a Framework for Hard Real-Time Applications. In: 31st Annual NASA Goddard Software Engineering Workshop (SEW 2007) (March 2007)

10. Lundqvist, K., Asplund, L.: A Formal Model of a Run-Time Kernel for Ravenscar. In: RTCSA 1999: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, p. 504. IEEE Computer Society, Washington, DC (1999)
11. Ober, I., Halbwachs, N.: On the Timed Automata-based Verification of Ravenscar Systems. In: Ada-Europe 2008. LNCS, vol. 5026, pp. 30–43. Springer, Heidelberg (to appear, 2008)
12. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
13. SAE. Architecture Analysis & Design Language (AS5506) (September 2004), `http://www.sae.org`
14. SEI. Open Source AADL Tool Environment (2006), `http://la.sei.cmu.edu/aadl/currentsite/tool/osate.html`
15. Sha, L., Klein, M.H., Goodenough, J.B.: Rate Monotonic Analysis for Real-Time Systems. Computer 26(3), 73–74 (1993)
16. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers 39(9), 1175–1185 (1990)
17. Vardanega, T., Zamorano, J., de la Puente, J.A.: On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels. Real-Time Syst. 29(1) (2005)