RCM Interface Grammar

Daniela Cancila and Tullio Vardanega

Abstract

This note presents the mathematical formalization of the model transformations developed around the RCM domain specific language (RCM is short for Ravenscar Computational Model) in the context of the ASSERT project partially funded by the European Commission in the 6th Framework Program (`www.assert-project.net`).

The mathematical basis for the present formalization of the RCM Interface Grammar was first published in the paper *"Composability for High-Integrity Real-Time Embedded Systems"* [1]. This technical note was written in complement to the paper *"Ensuring Correctness in the Specification and Handling of Non-Functional Attributes in High-Integrity Real-Time Embedded Systems"* by D. Cancila, R. Passerone, T. Vardanega and M. Panunzio, presently under review by the IEEE Transactions on Industrial Informatics.

I. Introduction

The purpose of the RCM Interface Grammar is to provide mathematically sound guarantees that the transformations that are applied to the Interface view (the user-level modeling space in the ASSERT development process) to automatically produce the the Implementation view do not incur semantic alteration.

To this end, the RCM Interface Grammar explicitly formalizes the whole set of model transformations that can take place between the Interface View and the Implementation View.

The RCM Interface Grammar introduces two formal languages: L_I for the Interface view and L_C for the Implementation view. The production and semantic rules on L_C define run-time components compliant, by construction, with the Ravenscar profile [2]. Model transformations are given by the inclusion of Language L_I into Language L_C. The guarantee that model transformations do not modify the semantics specified by the user in the Interface view is given by:

(1) Language L_I and Language L_C share the same syntax and semantics, which is technically possible because the Interface and the Implementation views share one and the same metamodel.
(2) The production rules are deterministic.

The two formal languages are introduced following the rules set forth in: A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman in "Compilers: Principles, Techniques, and Tools" [3]. Both formal languages have a syntax – given by terminal tokens, non-terminal tokens, and start symbols – and semantic. Production and semantic rules are given on Language L_C.

This note has the following structure. Section II introduces the RCM Interface Grammar providing an intuitive correspondence between the mathematical formalism and the user model. Section III and Section IV introduce the syntax and the semantics, respectively. Section V shows all production and semantic rules. Section VI discusses some properties and examples of the RCM Interface Grammar. The note ends by referring the reader to papers where we discuss applications of this approach, examples of use and the industrial evaluation of it.

II. Correspondence between the model and the RCM Interface Grammar

This section presents three tables that illustrate the correspondence between the RCM Interface Grammar and the user model.

D. Cancila is with CEA LIST, Commissariat à l'énergie atomique, Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués, France, daniela.cancila@cea.fr

T. Vardanega is with the Dipartimento di Matematica Pura e Applicata, University of Padova, Italy, tullio.vardanega@math.unipd.it

A. Historical perspective

The names of the syntactic tokens in the RCM Interface Grammar derive from the names of the operations first introduced by the HOOD design method and modeling language [4]. HOOD, short for *Hierarchical Object Oriented Design*, was the method of choice for the European Space Agency (ESA), in the decade between mid '80 and mid '90 (for examples of use see for instance the Columbus project [5]). HOOD was later supplanted by HRT-HOOD [6], which added real-time specific attributes and concerns to the original method and language. Both HOOD and HRT-HOOD considered the Ada programming language as the natural target of the code generated from the user model. The real-time semantics entailed HRT-HOOD in particular facilitated the adoption of the Ravenscar profile [2], a subset of Ada specifically tailored around the needs of high-integrity real-time embedded systems.

RCM is the (meta)modeling language for the Ravenscar profile. RCM enables users to specify Ravenscar-compliant components directly at modeling level without needing to code them.

label	**meaning of acronym**
$ASATC$	short for ASynchronous (Asynchronous) Transfer of Control
$ASER$	short for ASynchronous Execution Request
CER	short for Clocked Execution Request
$PAER$	short for Protected Asynchronous Execution Request
$PSER$	short for Protected Synchronous Execution Request
$OBCS$	short for OBject Control Structure
$OPCS$	short for OPeration Control Structure

TABLE I

ORIGINAL INTRODUCTION OF THE MODEL ELEMENTS IN HOOD

Models	**the RCM Interface Grammar**
(elementary) provided interface (in short) PI that contains only one method	token
specification of the method	attributes of a token
set of (elementary) PI in the Interface view	tokens in Language L_I
component in the Interface view	set of L_I tokens
set of (elementary) PI in the Implementation view	tokens in Language L_C
component in the Implementation view	set of L_C tokens given in the Right-Hand Side of a production rule
Model transformations between the Interface view and the Implementation view	inclusion between languages L_I and L_C

TABLE II

CORRESPONDENCE BETWEEN MODEL ELEMENTS AND FORMAL LANGUAGES

Models: Components	**the RCM Interface Grammar: Production Rules**
passive RCM run-time component, which provides data access with no synchronization guarantees	Right-Hand Side in Production Rule 7
protected RCM run-time component, which provides data access with synchronization guarantees	Right-Hand Side in Production Rule 6
cyclic *threaded* RCM run-time component, which have a clock-based release event and only perform a default operation	Right-Hand Side in Production Rule 1
cyclic *threaded* RCM run-time component, which can execute other operations than the default one (termed "modifiers")	Right-Hand Side in Production Rule 5
sporadic *threaded* RCM run-time component, which have a software-generated release event	Right-Hand Side in Production Rule 2
sporadic *threaded* RCM run-time component with modifiers	Right-Hand Side in Production Rule 4

TABLE III

CORREPONDENCE BETWEEN RUN-TIME COMPONENTS AND PRODUCTION RULES

Models: Operations	**the RCM Interface Grammar: Token**
elementary provided interface with *protected* method	token $PAER$, which identifies an operation that executes in the caller thread ($\mathcal{W} = 0$) with synchronization guarantees on data data ($\mathcal{S} = (h_w, 0)$) limited the current operation and not extended to the transitively called operations ($PAER.\tau = 0$).
elementary provided interface with *transactional* access guarantees	token $PAER$, same as above, but with synchronization guarantees extended transitively across all RIs involved in the execution of the PI ($\tau = RI^+$).
elementary provided interface with *passive* method	token $ASER$, which identifies an operation that executes in the caller thread ($ASER.\mathcal{W} = 0$) and provides no synchronization guarantee on access to data ($ASER.\mathcal{S} = (0, 0)$), and ($ASER.\tau = 0$).
provided elementary interface with *cyclic* method	token CER, which identifies a PI that is periodically invoked by the clock interrupt ($CER.\mathcal{W} = 1$, period T_c).
provided elementary interface with *sporadic* method	token $START$, which identifies a PI that is sporadically invoked by a software call ($START.\mathcal{W} = 1, T_t$) with T_t the Minimum separation time between successive executions.
provided elementary interface with *cyclic modifier* method	token $ASATC_c$, which denotes a non-default PI attached to a cyclic run-time component.
provided elementary interface with *sporadic modifier* method	token $ASATC_t$, which denotes a non-default PI attached to a sporadic run-time component.

TABLE IV

CORRESPONDENCE BETWEEN MODEL ELEMENTARY INTERFACES AND TOKENS

III. SYNTAX OF LANGUAGES L_I AND L_C

Syntax of Language L_I

start symbol = terminal token := $CER_1 \mid PAER \mid ASER$

Syntax of Language L_C

terminal token := $CER \mid PAER_{st} \mid PAER_0 \mid ASER_{st} \mid ASER_0 \mid ASATC_t \mid ASATC_c \mid START \mid PSER$

nonterminal token := $CER_1 \mid PAER \mid ASER \mid PAER^+ \mid ASER^+ \mid ASATC_t^+ \mid ASATC_c^+$

start symbol := $CER_1 \mid PAER \mid PAER^+ \mid ASER \mid ASER^+$

IV. SEMANTICS OF LANGUAGES L_I AND L_C

A. Semantics of the attributes

Tokens in Language $\mathcal{L}_I$ and in Language $\mathcal{L}_C$ share the same set of attributes

$$(\mathcal{P}, \mathcal{S}, \mathcal{W}, \mathcal{V}, \tau).$$

The values of the above attributes determine the inner parts of an RCM run-time component in terms of which metamodel elements they aggregate and of their specification.

The attributes are further refined as follows.

Functional Profile ($\mathcal{P}$), which designates the signature of the corresponding method and the further specification details.

Attribute $\mathcal{P}$: FUNCTIONAL PROFILE		
Attribute	=	**Meaning**
Parameter Profile	=	method signature
C_l	=	worst-case execution time of the PI, *exclusive* of the cost of all RI operations invoked by it
C_g	=	worst-case execution time of the PI *inclusive* of the cost of all RI operations invoked by it; C_g is *automatically calculated* on the transitive closure of the system model
Functional State	=	enumeration of the static variables operated upon by the PI operation
Invocations	=	the number of times that a single RI may be invoked in the execution of the PI operation
Ceiling	=	Priority ceiling value to apply on access to the functional state $_{st}$ of the component; this value is *automatically calculated*

Synchronization Protocol ($\mathcal{S}$), which is defined by the possible combination of values of the pair `(Access Type, Guard)`. $\mathcal{S}$ provides either no access protection to the functional state of the method, `(0,0)`; or exclusion synchronization $(h_w, 0)$; or avoidance synchronization, (h_w, G), subject to Boolean guard G.

Concurrent Weight ($\mathcal{W}$), which can only value zero or one, to signify the absence or respectively the presence (the need for) of a THREAD in the designated run-time component. When $\mathcal{W} = 0$ the caller may directly operate on the resources of the callee. Otherwise, the callee is doted with an own thread of control that executes the invoked service on behalf of the caller (in a manner akin to the classical client-server model). The THREAD designated by $\mathcal{W} = 1$ can have specific attributes such as deadline or priority.

Attribute $\mathcal{W}$: CONCURRENT WEIGHT		
Attribute	**=**	**Meaning**
$\mathcal{W} = 0$	=	The caller may directly operate on the resources of the callee
$\mathcal{W} = 1$	=	the callee is doted with an own thread of control that executes the invoked service on behalf of the caller.
Functional Enumeration μ^{ϵ}	=	Finite and statically defined list of all operations that a single THREAD may execute, in addition to the default operation
tag_i μ_i^{ϵ}	=	Single operation in the functional enumeration, termed *modifier*.
Priority Pr	=	Urgency of the THREAD
Deadline D	=	Latest completion time of every single operation executed by the THREAD
Period T_c	=	Period for cyclically activated operations
Minimum Interarrival Time T_t	=	Minimum separation time between successive sporadic activations

Visibility ($\mathcal{V}$), to the RI of other components in the system. $\mathcal{V}$ can evaluate to: `public` (the method is visible to all components); `private` (the method is only visible to the RI of its own component); `restricted` (the method is visible to some specified components).

Transactional Access (τ), to a resource, when $\tau = RI^{+}$, which implies that the corresponding method shall be executed with transactional guarantees over the invocation of the RI specified in τ. $\tau = 0$ if no such guarantee is required.

Notation: Symbol '*' in the following denotes that a literal can take any meaning specified by the semantics of the attribute. For example $PAER.\mathcal{V}.*$ means that literal $PAER$ can take public, private or restricted visibility.

Summary: attributes of a provided elementary interfaces		
Elementary Interface	**=**	**Attributes**
$PI.\{\mathcal{P}, \mathcal{S}, \mathcal{W}, \mathcal{V}, \tau\}$	=	$PI.\mathcal{P} ::= PI.\{*, (C_l, C_g), *, -, \texttt{Ceiling}\}$ where C_g is calculated and not set by the user. The `Ceiling` attribute is set only on all PI that have a non-void protocol (that is, $PI.\mathcal{S} \neq (\emptyset, \emptyset)$) and it is calculated by model transformation and not set by the user. $PI.\mathcal{S} = PI.\{*\}$ $PI.\mathcal{W} = PI.\{*\}$ If $\mathcal{W} = \emptyset$, then the PI is immediate, else it is deferred. Only a deferred PI has Deadline, Priority, Period or Minimum Interarrival Time attributes. Every single nominal activation PI $(1_{t\|c})$ belongs in given a functional enumeration denoted $(\epsilon, \{\mu_1, \ldots, \mu_k\})$, where: ϵ is an identifier unique per run-time component, which denotes a specific PI that designates a nominal activation; and $\{\mu_1, \ldots, \mu_k\}$ is a set of deferred PI each of which denotes a modifier to the nominal activation specified by the corresponding unique identifier. $PI.\mathcal{V} = PI.\{*\}$ $PI.\tau = PI.\{*\}$ when $\tau \neq 0$ then τ designates an ordered list of RI $(RI_1; \ldots; RI_n)$. Every RI may be satisfied by a PI equipped with a matching parameter profile and with either a horizontal protocol or void protocol *and* no other caller in the system than that particular RI. Furthermore, the first and the last RI have a private visibility and, hence, they are satisfied by the PI by the run-time component that provides the transactional access.

B. Semantics of Tokens in Language L_I

CER_1 ***semantics***	
$CER_1.\mathcal{P} =$	$\{0, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$
$CER_1.\mathcal{W} =$	$1_c, \mu^\epsilon, Pr, T_c, D$
$CER_1.\mathcal{S} = (h_w, G)$	$CER_1.\mathcal{V} = pr \quad CER_1.\tau = 0$

Literal CER_1 specifies the default operation periodically executed by the THREAD ($\mathcal{W} = 1$) associated with that PI.

$ASER$ ***semantics***	
$ASER.\mathcal{P} =$	$\{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\} \mid \{*, (C_l, C_g), \emptyset, -, \texttt{Ceiling}\}$
$ASER.\mathcal{W} = 0$	$ASER.\mathcal{S} = (0, 0)$
$ASER.\mathcal{V} = *$	$ASER.\tau = 0$

Literal $ASER$ denotes an operation executed by the caller ($\mathcal{W} = 0$) with no guarantee of integrity in case of concurrent calls to that PI ($\mathcal{S} = (0, 0)$).

$PAER$ ***semantics***	
$PAER.\mathcal{P} =$	$\{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$
$PAER.\mathcal{W} =$	$0 \mid 1_t, \mu^\epsilon, Pr, T_t, D \mid 1_v, \mu_i^\epsilon, Pr, T_{c\|t}, D$
if $PAER.\mathcal{W} = 0$	$PAER.\mathcal{S} = (h_r, 0) \mid (h_w, 0)$, $PAER.\mathcal{V} = *$, $PAER.\tau = 0 \mid RI^+$
if $PAER.\mathcal{W} = 1_t, \mu^\epsilon, Pr, T_t, D \mid 1_v, \mu_i^\epsilon, Pr, T_{c\|t}, D$	$PAER.\mathcal{S} = (h_w, G)$ $PAER.\mathcal{V} = *$ $PAER.\tau = 0$

Literal $PAER$ can assume different meanings. When $\mathcal{W} = 0$, literal $PAER$ denotes an operation executed by the caller with exclusion synchronization guarantees in case of concurrent calls to that PI ($\mathcal{S} = (h_w, 0)$). When $\mathcal{W} = 1$, literal $PAER$ signifies that the caller operation delivers an execution request to a THREAD that resides on the side of the callee. In that case, literal $PAER.\mathcal{W} = 1_c|1_t$ may take one of the following three interpretations:

- causes the THREAD designated by the associated CER literal to execute the $PAER$ operation at its next activation instead of the default operation ($PAER.\mathcal{W} = 1_v, \mu_i^\epsilon, Pr, T_c, D$)
- delivers an activation event to an associated sporadic THREAD, which causes it to execute its default operation ($PAER.\mathcal{W} = 1_t, \mu^\epsilon, Pr, T_t, D$)
- delivers an activation event to an associated sporadic THREAD, which causes it to execute the $PAER$ operation instead of the default one ($PAER.\mathcal{W} = 1_v, \mu_i^\epsilon, Pr, T_t, D$).

C. Semantics of Tokens in Language L_C

In this section we only introduce the semantics of tokens which are in $\mathcal{L}_C$ but not in $\mathcal{L}_I$.

$PSER$ ***semantics***	
$PSER.\mathcal{P} =$	$\{\star, C_l, \text{State}, -, \texttt{Ceiling}\}$
$PSER.\mathcal{W} =$	$1_{c\|t}, -, Pr, T_{c\|t}, D$
$PSER.\mathcal{S} = (v, G)$	$PSER.\mathcal{V} = pr \quad PSER.\tau = 0$

Literal $PSER$ denotes an operation executed by the THREAD of the callee ($\mathcal{W} = 1, \mathcal{V} = pr$) and specifies the state-based condition, expressed by Boolean guard G ($\mathcal{S} = (v, G)$) on which that execution is dependent.

The semantics of CER is the same as that of CER_1 in $\mathcal{L}_I$. The only reason to syntactically distinguish between CER and CER_1 is to avoid recurrence in production rules.

Literal $ASER$ specifies the semantics of $ASER_{st}$ and $ASER_0$, with the former operating on functional

state $_{st}$ and the latter operating on none (hence as a pure function).

$PAER$ ***semantics***	
$PAER_{st}$	$PAER_{st}.\mathcal{P} = \{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$, $PAER_{st}.\mathcal{S} = (h_w, 0)$, $PAER_{st}.\mathcal{W} = 0$, $PAER_{st}.\tau = 0$, $PAER_{st}.\mathcal{V} = *$
$PAER_{\tau}$	$PAER_{\tau}.\mathcal{P} = \{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$, $PAER_{\tau}.\mathcal{S} = (h_w, 0)$, $PAER_{\tau}.\mathcal{W} = 0$, $PAER_{\tau}.\tau = RI^{+}$, $PAER_{\tau}.\mathcal{V} = *$
$START$	$START.\mathcal{P} = \{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$, $START.\mathcal{S} = (h_w, G)$, $START.\mathcal{V} = *$, $START.\tau = 0$, $START.\mathcal{W} = 1_t, Pr, T_t, D$
$ASATC_t$	$ASATC_t.\mathcal{P} = \{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$, $ASATC_t.\mathcal{S} = (h_w, G)$, $ASATC_t.\mathcal{V} = *$, $ASATC_t.\tau = 0$, $ASATC_t.\mathcal{W} = 1_v, \mu_i^{\epsilon}, Pr, T_t, D$
$ASATC_c$	$ASATC_c.\mathcal{P} = \{*, (C_l, C_g), \text{State}, -, \texttt{Ceiling}\}$, $ASATC_c.\mathcal{S} = (h_w, 0)$, $ASATC_c.\mathcal{V} = *$, $ASATC_c.\tau = 0$, $ASATC_c.\mathcal{W} = 1_v, \mu_i^{\epsilon}, Pr, T_c, D$

Literal $PAER$ is specialized in: literal $PAER_{st}$, which endows the operation with exclusion synchronization guarantees on access to functional state $_{st}$; $PAER_{\tau}$, which guarantees transactional access to the operation and its functional state $_{st}$ inclusive across the execution of all of the RI invoked by it; $START$, which delivers an activation event to the associated sporadic THREAD, which causes it to execute its default operation; $ASATC_t$, which delivers an activation event to the associated sporadic THREAD, which causes it to execute the $PAER$ operation instead of the default one; $ASATC_c$, which does the same as $ASATC_t$ but only for a cyclic THREAD.

V. Production and Semantic Rules

Literals in $\mathcal{L}_I$ and $\mathcal{L}_C$ have the same set of attributes. The attributes set in the LHS of a production rule determine the RHS that proceeds from it. This is for example the case with Rules (2,4,5,6) below where the semantic specialization of the LHS $PAER$ literal resolves to a distinct RHS rule deterministically.

Rule (1): cyclic run-time component	
Production	*Semantics*
$CER_1 \rightarrow (CER, PSER)$	$CER ::= CER_1$

Rule (**1**) defines the nominal behavior of a cyclic run-time component. CER_1 delegates all its attributes to literal CER. $PSER$ is attached to a Boolean guard (off an avoidance synchronization attribute) which is opened by the invocation of CER.

Rule (2): sporadic run-time component	
Production	*Semantics*
$PAER \rightarrow (START, PSER)$	$PAER.\mathcal{S} = (h_w, G)$, $PAER.\mathcal{S} = (h_w, G)$, $PAER.\mathcal{W} = \{1_t\}$ and $START ::= PAER$

Rule (**2**) defines the nominal behavior of a sporadic run-time component. The $PAER$ literal delegates all attributes to $START$ literal, which is *always* paired with one literal $PSER$, much like ratified by Rule (**1**).

Rule (3a,b): modifiers	
Production	*Semantics*
(**3a**) $ASATC_t^{+} \rightarrow ASATC_t \vert ASATC_t, ASATC_t^{+}$	(**3a**) $ASATC_t := ASATC_t^{+}$
(**3b**) $ASATC_c^{+} \rightarrow ASATC_c \vert ASATC_c, ASATC_c^{+}$	(**3b**) $ASATC_c := ASATC_c^{+}$

Rules (**3a**) and (**3b**) define modifiers for sporadic and cyclic run-time components, respectively.

Rule (4): sporadic run-time component with modifiers	
Production	*Semantics*
$PAER^{+} \rightarrow (ASATC_t^{+}, START, PSER)$	$PAER^{+}.\mathcal{S} = (h_w, G)$, $PAER^{+}.\mathcal{W} = 1_v, T_t$ and $ASATC_t^{+} := PAER^{+}$

Rule (**4**) states that any number of modifiers can be specified for a sporadic run-time component only

after the nominal behavior ($START$) has been specified.

Rule (5): cyclic run-time component with modifiers	
Production	*Semantics*
$PAER^+ \rightarrow (ASATC_c^+, CER, PSER)$	$PAER.\mathcal{S} = (0, G)$, $PAER.\mathcal{W} = 1_v, T_c$ and $ASATC_c^+ := PAER^+$

Rule (**5**) does the same as Rule (**4**) to cyclic run-time components.

Rules (6a,b): protected run-time component	
Production	*Semantics*
(6a) $PAER^+ \rightarrow (PAER_{st}) \mid (PAER_{st}, PAER^+)$	**(6a)** $PAER_{st} ::= PAER^+$ and $PAER^+.\mathcal{S} = (h_r, 0) \mid (h_w, 0)$, $PAER^+.\mathcal{W} = 0$ and $PAER^+.\tau = 0$
(6b) $PAER^+ \rightarrow (PAER_{\tau}) \mid (PAER_{\tau}, PAER^+)$	**(6b)** $PAER_{\tau} ::= PAER^+$ and $PAER^+.\mathcal{S} = (h_r, 0) \mid (h_w, 0)$, $PAER^+.\mathcal{W} = 0$ and $PAER^+.\tau = RI^+$

Rule (**6a**) defines a protected run-time component, which warrants mutual exclusion on access to the logical resources (termed "functional state", subscript $_{st}$) by the component in the execution of its provided interfaces. Rule (**6b**) stipulates that it can also provide for transactional access guarantees.

Rules (7a,b): passive run-time component	
Production	*Semantics*
(7a) $ASER_{st}^+ \rightarrow (ASER_{st}) \mid (ASER_{st}, ASER^+)$	**(7a)** $ASER_{st} ::= ASER_{st}^+$
(7b) $ASER_0^+ \rightarrow (ASER_0) \mid (ASER_0, ASER^+)$	**(7b)** $ASER_0 ::= ASER_0^+$

Rules (**7a**) and (**7b**) define a passive run-time component that may contain any number of LHS literals so long as they either operate on one and the same functional state or they have no functional state.

VI. Properties

Proposition 1 argues that any given token in L_C is realized by exactly one RCM run-time component.

Proposition 1 (Determinism): The grammar on L_C is unambiguous.

Proof: By construction. We first prove that a token in L_C belong to only one LHS. Then we prove that each RHS gives a unique set of terminal tokens.

To prove the first item, we constructively compare a given token of a LHS with LHS tokens, which belong to the remaining production rules. We have therefore:

Token CER_1 only belongs to Rule (1).

Token $PAER$ such that $PAER.\mathcal{W} = \{1_t\}$ only appears in Rule (2).

Token $PAER^+$ such that $PAER^+.\mathcal{W} = 1_v, T_t$ and $PAER^+.\mathcal{S} = (h_w, G)$ only appears in Rule (4).

Token $PAER^+$ such that $PAER^+.\mathcal{W} = 1_v, T_c$ and $PAER.\mathcal{S} = (0, G)$ only appears in Rule (5).

Token $PAER^+$ such that $PAER.\mathcal{W} = 0$ and $PAER.\tau = 0$ only appears in Rule (6a).

Token $PAER^+$ such that $PAER.\mathcal{W} = 0$ and $PAER.\tau = RI^+$ only appears in Rule (6b).

Token $ASATC_t^+$ only appears in Rule (3a).

Token $ASATC_c^+$ only appears in Rule (3b).

Token $ASER_{st}^+$ only apperas in Rule (7a).

Token $ASER_0^+$ only apperas in Rule (7b).

To prove the second item, we constructively analyse each production rule.

Rule (1) gives a unique set of terminal tokes. Likewise for Rule (2).

Rule (4) allows designers to introduce as many modifiers as they want. To this end, Rule (4) contains the non-terminal token $ASATC_t^+$ in the RHS. Token $ASATC_t^+$ only appears in Rule (3a). This rule generates a finite number of modifiers. Rule (4) is always applied to a statically-bounded scenario. Hence, the number of modifiers in a system is always statically fixed. Therefore, the effects of Rule (3a) are always bounded. Likewise for Rule (5) which implicitly uses Rule (3b).

Rule (6a) generates a finite number of tokens, as many as designer wants. Likewise for Rule (6b), Rule (7a) and Rule (7b). ■

Proposition 2: The production and semantic rules in the RCM Interface Grammar only generate run-time components compliant to RCM.

Proof: By construction. RCM has three types of run-time components: passive, protected and threaded.

Rules 7a and 7b produces a passive run-time component.

Rules 6a and 6b produces a protected run-time component.

Rules 1 produces a threaded run-time component, with a clock-based release event, termed *cyclic*.

Rules 5 produces a cyclic threaded run-time component, with finite set of modifiers.

Rules 2 produces a threaded run-time component, with a software-generated release event, termed *sporadic*.

Rules 4 produces a sporadic theraded run-time component, with a finite set of modifiers. ■

Proposition 3: $L_I \subset L_C$.

Proof: By construction. Each token in L_I is also a token in LHS of Language L_C. Moreover, by design, L_I and L_C share the same set of semantic attributes. ■

Proposition 1 and 3 enable us to (automatically) realize any single method of a component in the Interface view in a given RCM run-time component in the Implementation view with the guarantee to preserve the intended semantics.

A. *Examples*

Figure 1 and Figure 2 show two examples of the production and semantic rules. The designer specifies a method in the Interface view. It corresponds to a token in LHS. In RHS, the transformation automatically generates the RCM run-time component that realizes the method. RCM run-time components are then analyzed for schedulability and the result is back propagated in the LHS. (Extended discussion given in the paper "Ensuring Correctness in the Specification and Handling of Non-Functional Attributes in High-Integrity Real-Time Embedded Systems" by D. Cancila, R. Passerone, T. Vardanega and M. Panunzio.)

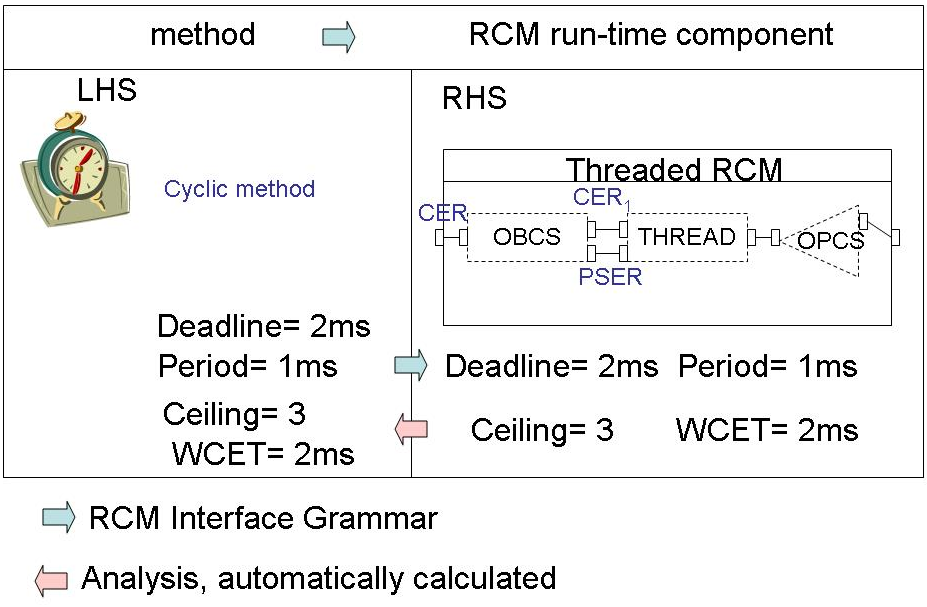

Fig. 1. Graphical representation of an example of use of Rule (1)

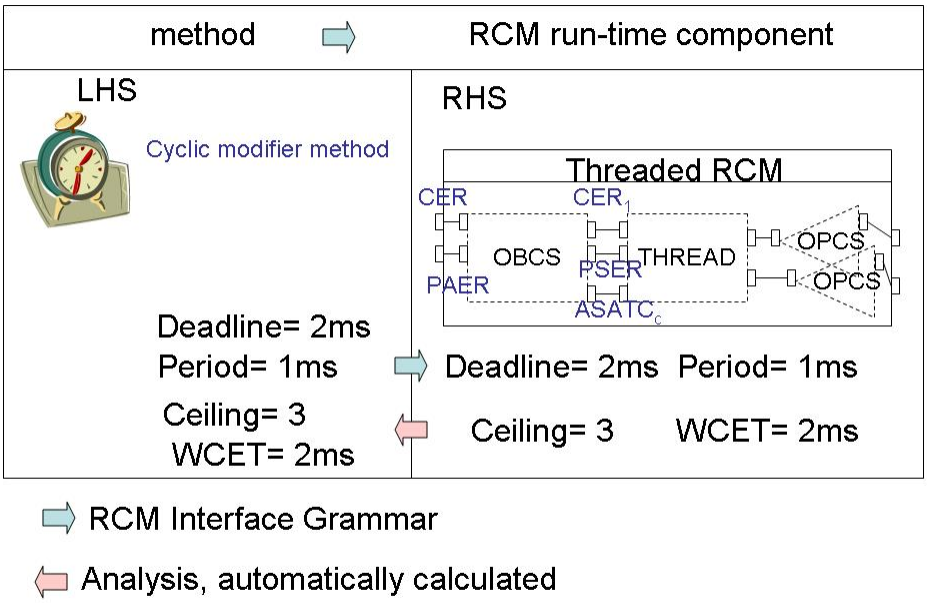

Fig. 2. Graphical representation of an example of use of Rule (5)

VII. Application, Examples and Industrial Evaluation

The RCM Interface Grammar was defined as part of the research work carried out in the ASSERT project partially funded by the European Commission in the 6th Framework Program (`www.assert-project.net`).

During that project, a team at the University of Padova also developed a prototype tool for the design of real-time embedded applications based on the RCM Interface Grammar. The prototype tool, and then implicitly its formal foundation via the RCM Interface Grammar, were successfully experimented in two industrially-relevant case studies. The proceedings of the industrial experiments example were discussed in ASSERT deliverables and in [1], among others. Industrial evaluation of the RCM Interface Grammar was very encouraging. A critical analysis of that industrial evaluation is reported in "Ensuring Correctness in the Specification and Handling of Non-Functional Attributes in High-Integrity Real-Time Embedded Systems" by D. Cancila, R. Passerone, T. Vardanega and M. Panunzio.

References

[1] D. Cancila, R. Passerone, and T. Vardanega, "Composability for high-integrity real-time embedded systems," in *Proceedings of the First Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 08)*, Barcelona, Spain, November 30, 2008.

[2] A. Burns, B. Dobbing, and T. Vardanega, "Guide to the Use of the Ada Ravenscar Profile in High Integrity Systems," University of York (UK), Tech. Rep. YCS-2003-348, 2003, http://www.cs.york.ac.uk/ftpdr/reports/YCS-2003-348.pdf.

[3] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tool*. Addison-Wesley, 1986.

[4] HOOD Technical Group, *HOOD: An industrial approach for software design*. HOOD User's Group - J.P. Rosen, 1997.

[5] C. M. Space, "The Manned Space and Microgravity Programmes," http://www.esa.int/esapub/br/br114/br114man.htm, last visit in February 2006.

[6] A. Burns and A. Wellings, *HRT-HOOD A Structural Design Method for Hard Real-Time Ada Systems*. University of York (UK), Elsevier, 1995.